Documentation Programmeur

Projet : p2208, Développement d'un Capteur IoT LoraWAN

Table des matières

Prérequis

Pour pouvoir manipuler le firmware de Dragino pour notre application, il y a besoin de plusieurs prérequis, en voici la liste.

- Firmware du Dragino LSN50 : https://github.com/dragino/LoRa_STM32
- Datasheet de la sonde à intégrer au firmware
- Keil µVision IDE : http://wiki.dragino.com/xwiki/bin (suivre l'explication de cette partie de la documentation de Dragino, notamment compile instruction)

Création d'un nouveau mode

Présentation

Le firmware du Dragino LSN 50 repose sur un algorithme spécifique, présenté en figure 1, qui va opérer sur plusieurs actions. Ici, celle qui nous intéresse est entourer en bleu et correspond au mode de fonctionnement.

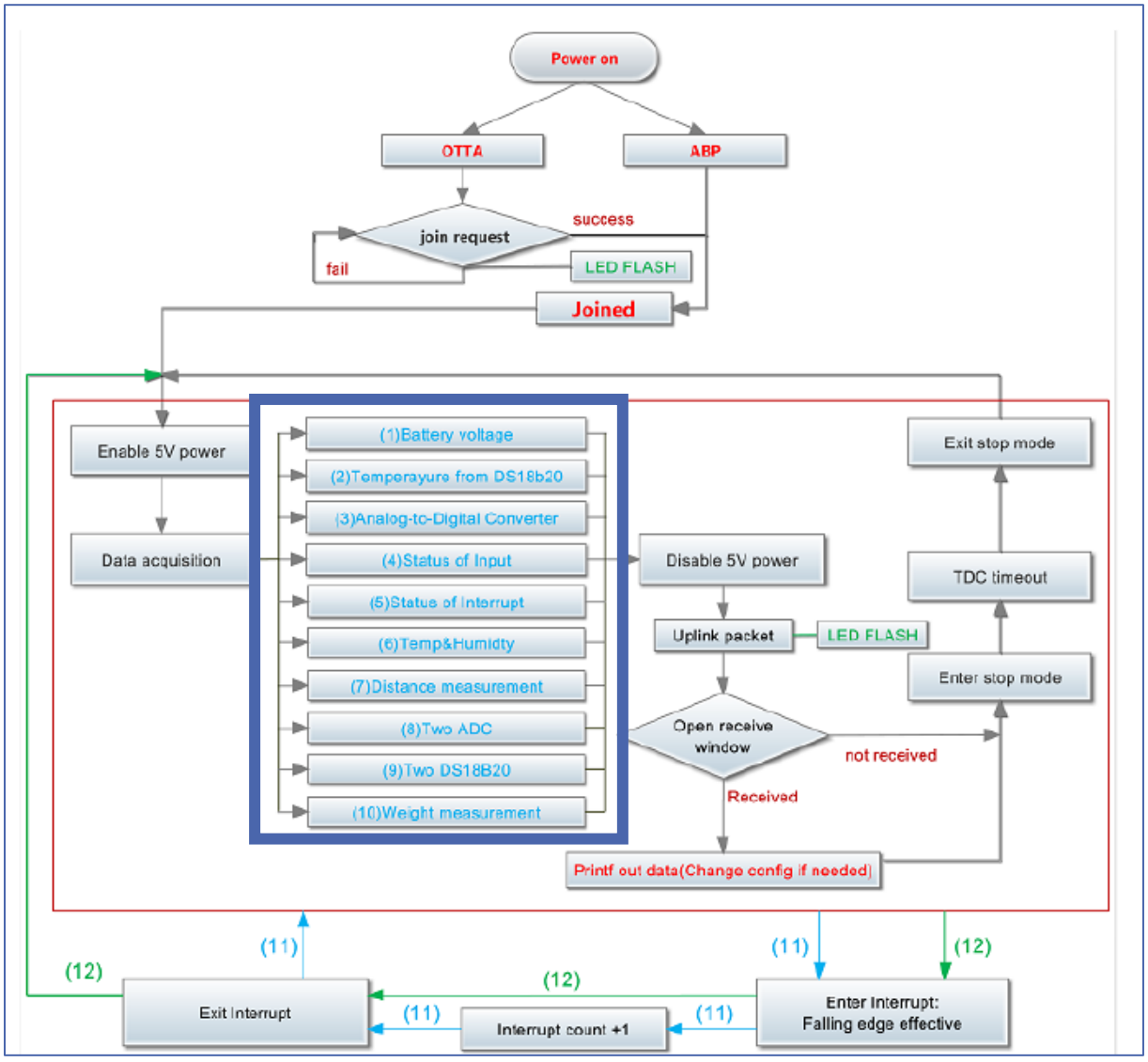

Figure 1: Diagramme de fonctionnement du Dragino LSN50 (version 1.7 du firmware)

En effet, on remarque que dans cet algorithme, il y a le choix entre 10 modes de fonctionnement natif, dans la version 1.7 du firmware. Ces modes de fonctionnements

sont décrits dans le manuel utilisateur du constructeur et ils font des actions spécifiques.

Le but de cette partie est de montrer comment créer un nouveau mode de fonctionnement dans le firmware du Dragino LSN50 afin qu'il puisse exécuter un code qu'on aura écrit.

Ajout du mode dans le fichier at.c

Cette action est à faire à chaque fois que l'on veut ajouter un nouveau mode. Elle permet d'identifier le mode que nous créons et de pouvoir rentrer dans ce mode-ci.

L'une des premiers choses à faire c'est de choisir le numéro de son mode. Plusieurs choses : vu que c'est un unsigned int on ne peut pas dépasser 255 ET comme Dragino met à jour son firmware, des modes vont s'ajouter (1.7.2, 9 modes -> 1.8, 11 modes), Ainsi il faut prendre un numéro élevé (ex : 40 pour ce projet).

Une fois cela fait il faut modifier le fichier qui s'occupe des commandes AT pour modifier celle qui gère les modes (pour pouvoir rentrer dans le nouveau mode). Ici on ajoute seulement le mode 40 donc on doit rentrer dans le IF de la commande AT « mode » si le mode est parmi les modes déjà créer OU si le mode est 40.

```
if (((workmode>=1)&&(workmode<=9))||(workmode==40))
{
  mode=workmode;
  PPRINTF("Attention:Take effect after ATZ\r\n");
}
```

Figure 2: Intégration du mode 40 au program

Malheureusement, cette partie est à refaire à chaque fois qu'il y a une mise à jour du code source par le constructeur, **seulement l'on souhaite passer à la nouvelle version.** Par exemple, si on veut passer de la version 1.7 à 1.8, il faudra copier notre code pour le mettre dans le nouveau at.c **s'il est modifié.** Ici cela ne concerne seulement les valeurs numériques que peut prendre le mode.

Modification dans main.c

Le deuxième changement à faire est dans main.c, c'est ici que les trames LoRaWAN sont construite, et qu'on peut exécuter le code spécifique à notre mode. On ajoute ainsi dans la fonction « send », un nouveau IF pour le nouveau mode (mode 40 pour notre

projet). Dans ce IF on décrit la trame que l'on veut et on suit les autres écritures. Dans la figure 3, par exemple, on envoie juste l'entier 99 dans la trame.

```
else if(mode==40)
{

  AppData.Buff[i++]=(int) 99;      //test

}
```

Figure 3: Modification dans main.c

Maintenant, nous avons un nouveau mode spécial pour notre application, et nous allons pouvoir envoyer en LoRaWAN la trame que nous souhaitons, avec les données que nous voulons. Tout cela, sans interférer avec le code de base donc en gardant la compatibilité avec les autres modes.

Intégration d'un nouveau capteur

Recherche

La première étape dans la démarche pour développer un driver pour un nouveau capteur est de se renseigner sur le protocole de communication du capteur (SPI, I2C, Serial, …). C'est le plus important car c'est la base du driver. Pour cela, la datasheet du capteur doit fournir cette information. Dans notre cas il s'agit de l'I2C.

A partir du protocole de communication du capteur, il faut maintenant regarder dans le code source du dragino (/Drivers/BSP/Components) s'il intègre ou non déjà un capteur utilisant ce protocole. Cela va permettre d'avoir un exemple pour construire notre driver. Dans notre projet c'est le cas, en effet, la sonde SHT 31 pris en charge par le firmware du dragino fonctionne en I2C et s'apparente bien à l'intégration de la sonde HYT939.

Ensuite, on regarde sur internet, s'il y a déjà un driver pour notre capteur ou non. Généralement, il existe un code Arduino ou un code en c pour notre sonde. On va pourvoir s'inspirer de ce code avec celui déjà existant pour créer notre propre driver pour notre sonde.

Manipulation

Actuellement, le code du driver déjà existant, SHT31.c pour nous, ne marche pas avec notre sonde, ce qui est normal car ces deux sondes ne se contrôle pas de la même manière et ne communique pas avec les mêmes adresse I2C. Cependant, les actions afin de récupérer une nouvelle donnée sont les mêmes : Mesurement Request et Data Fetch. On reprend ainsi la structure du driver du SHT31 et on le modifie pour nos besoins. Pour cela, on s'inspire du code Arduino que l'on a à notre possession (voir figure 4).

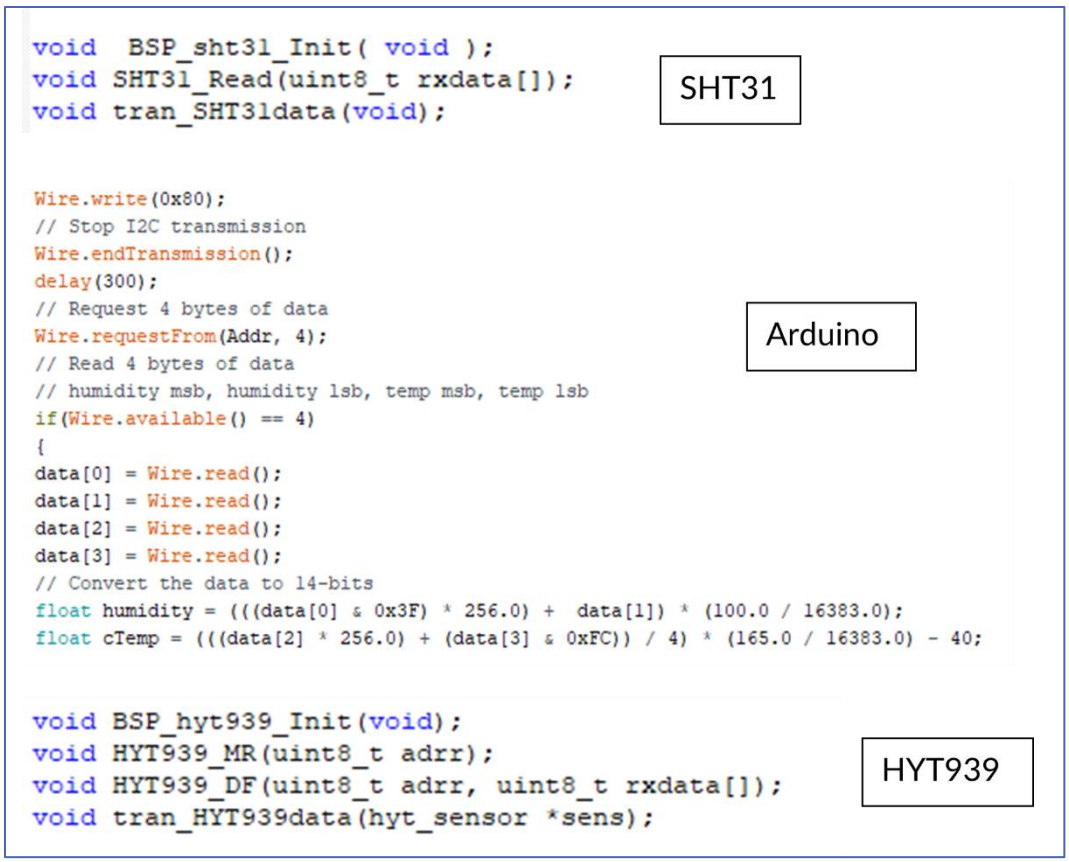

Figure 4: Création à partir des recherches de la structure du driver

Une fois cela fait, on peut écrire le driver pour une sonde HYT939 avec la datasheet du composant et du code Arduino encore une fois.

Bsp.c

Dans la construction de l'application, les développeurs du Dragino LSN50 ont intégrer la gestion de tous les drivers créer au fichier bsp.c. Ce sont les fonctions de ce fichier qui sont appelées dans la main de l'application et qui vont permettre d'initialiser les drivers et de les utiliser. Ainsi, on doit ajouter dans ce fichier les parties de code pour

faire cela avec notre nouvelle sonde. Tout d'abord dans le bsp.h où on ajoute les variables que l'on veut transmettre au main.c, voir figure 5.

```
typedef struct{

  uint8_t   in1;/*GPIO Digital Input 0 or 1*/

  float temp1;//DS18B20-1

  float temp2;//DS18B20-2

  float temp3;//DS18B20-3

  float oil;  //oil float

  float ADC_1; //ADC1

  float ADC_2;  //ADC2

  float temp_sht;

  float hum_sht;

  float temp_hyt;

  float hum_hyt;

  uint16_t illuminance;

  uint16_t distance_mm;

  uint16_t distance_signal_strengh;

  int32_t Weight;

  hyt_sensor hyt_sens[10];
  /**more may be added*/
} sensor_t;
```

Figure 5: ajout des variables dans la structure qui ai appelé dans le main.c

Puis dans le fichier bsp.c avec la fonction init et read que nous voyons juste après. Nous avons donc la possibilité de passer les valeurs de notre sonde dans le main pour les envoyés dans la trame Lora. Il faut maintenant dire comment elles sont récupérées, avec notre driver.

```
else if (mode == 40)
{
  HAL_I2C_MspInit(&I2cHandle40);
  tran_HYT939data(void);
}
```

Figure 6: ajout dans la fonction read dans bsp.c

Voici pour la fonction read, encore une fois, on ajoute le code pour le mode de notre application, ici c'est le 40. A savoir

que la fonction appelée ici, tran_HYT939data écris dans la valeur de température et d'humidité les valeurs mesurées.

```
else if (mode == 40)
{
        HAL_I2C_MspDeInit(&I2cHandle40);
        BSP_hyt939_Init();
}
```

Ici c'est pour la fonction d'initialisation.

Figure 7: ajout dans la fonction init dans bsp.c

Ainsi, on peut maintenant envoyer les valeurs mesurées par les sondes sur le réseau LoRaWAN.

Prise en charge de plusieurs sondes de même référence

Maintenant, on aimerait ne pas lire qu'une seule sonde mais plusieurs à chaque cycle de réveille. On va devoir faire quelque modification afin de faire cela.

Différencier les sondes

La première chose à faire est de pouvoir différencié les sondes une à une dans le code, afin que chaque valeurs lus et envoyés soit bien celle qui est associé à la bonne sonde.

Pour faire cela, chaque sonde aura un attribut en plus que ceux qu'il a déjà, pour nous sa température et son humidité. Pour notre projet, ce sera l'adresse I2C de la sonde, mais ce peut être un numéro d'identification.

Et, maintenant qu'une sonde est différenciée d'une autre, on peut créer « l'objet » sonde ou capteur associé au notre. La notion d'objet n'étant pas présente en C on utilise une structure :

```
typedef struct{

  uint8_t adrr;

  float temp;

  float hum;

  uint8_t gain;

  uint8_t offset;

} hyt_sensor;
```

Figure 8: structure pour les attribues de la sonde HYT939

Ainsi on y retrouve la température et l'humidité associé à chaque sonde grâce à leur adresse. Il y a aussi une valeur de gain et un offset pour la correction linéaire pour l'humidité.

Modification de structure et fonction déjà existante

Nous avons vu figure 5 que les données étaient passées à la fonction main via une structure regroupant plusieurs types de capteur. Nous avions ajouté nos valeurs, cependant, cela n'est plus d'actualité dans notre cas, car nous voulons les données de plusieurs sondes identifiées. Pour cela on créer un tableau avec chaque sonde différencié (voir figure 10).

En plus de cela, il faut maintenant modifier le driver de notre fonction pour qu'il communique avec chacune des sondes. Maintenant on utilisera des pointeurs sur chaque sonde spécifique. Un exemple ci-dessous, figure 9.

```
void tran_HYT939data(hyt_sensor *sens);
```

Figure 9: modification dans le driver

On voit dans cet exemple qu'on doit spécifier la sonde avec laquelle on travail pour ensuite savoir à quelle adresse aller chercher les mesures et où les stocker.

```
typedef struct{
  uint8_t   inl;/*GPIO Digital Input 0 or 1*/
  float temp1;//DS18B20-1
  float temp2;//DS18B20-2
  float temp3;//DS18B20-3
  float oil;  //oil float
  float ADC_1; //ADC1
  float ADC_2;  //ADC2
  float temp_sht;
  float hum_sht;
  hyt_sensor hyt_sens[10];
  uint16_t illuminance;
  uint16_t distance_mm;
  uint16_t distance_signal_strengh;
  int32_t Weight;
  /**more may be added*/
} sensor_t;
```

Figure 10: modification de la structure de capteur

Alimentation des sondes avec une entrée sortie

Pour alimenter un appareil avec une entrée-sortie (IO), la première chose à regarder est si cela est possible. Les valeurs de consommation sont données dans les datasheets, de même pour la valeur que peut fournir un IO du Dragino LSN 50V2 (16mA).

Ensuite, il faut choisir le pin qui va alimenter les sondes. Encore une fois, la datasheet et le manuel utilisateur du Dragino LSN50 fournie des informations sur les pins, leur fonction et leurs caractéristiques.

Une fois cela fait, il suffit de suivre un exemple dans le firmware ou de se référer aux commentaires dans le fichier sources de la HAL pour savoir comment faire.

Premièrement, il faut initialiser l'IO, voir figure 11. Ici, on dit qu'il est en sortie, avec une résistance de pull-up interne.

```
GPIO_InitTypeDef GPIO_InitStruct={0};

GPIO_InitStruct.Pin = GPIO_PIN_10;
GPIO_InitStruct.Mode  = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull  = GPIO_PULLUP;
GPIO_InitStruct.Speed = GPIO_SPEED_HIGH;

HW_GPIO_Init( GPIOA, GPIO_PIN_10, &GPIO_InitStruct );
```

Figure 11: initialisation d'un IO

Ensuite, on peut activer ou non la sortie, comme dans l'exemple de la figure 12.

```
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_RESET);
HAL_Delay(50);
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_SET); //
HAL_Delay(50);
```

Figure 12: Set et Reset d'une entrée sortie

Ajout de commande AT

Ce point a été détaillé dans la note d'application nommé ainsi, qui peut se retrouver joint dans les documentations.

Optimisation de la consommation

Ce point a été détaillé dans la note d'application nommé ainsi, qui peut se retrouver joint dans les documentations.