User Guide
FSK_Stack

Table des matières

1. Présentation de la pile de communication

La pile *FSK_Stack* ressemble un peu à la pile Xbee très très simplifié. Le système fonctionne avec un émetteur/récepteur FSK half duplex (RT606) à une fréquence de 433MHz. Il n'y a pas de notion de réseau. Toutefois, chaque trame physique est codée par un préambule qui permet de filtrer d'autres transmissions (alarme, ouverture voiture, etc...). Ce préambule est « ##### ». On peut donc dire que l'ensemble des modules communiquant sont « dans un réseau » identifié par ce préambule.

Chaque module possède son adresse, (son « my » pour comparer avec le Xbee), codée sur 8bits. La notion de broadcast existe aussi. Il s'agit de l'adresse 0xFF. On considérera ces adresses comme étant des adresses MAC.

Pour durcir la communication, un checksum est fait.

Enfin, émission comme réception se font sur la connaissance de la longueur du message à envoyer. Celui ci est limité mais configurable.

Résumé des fonctionnalités de la pile :

- FSK 433MHz Half Duplex,
- Appartenance à un réseau identifié par ##### en début de communication,
- Adresses « MAC » pour chaque device (8 bits), 255 adresses utilisables,
- Broadcast @ = 255,
- Checksum.

1.1. Accès médium

Le module FSK peut être soit en émission, soit en réception. Par défaut, le système est en écoute du canal. C'est l'état par défaut dans la machine à états qui contrôle la pile.

En réception, si la séquence « ##### » est détectée (Header detected), le système quitte son état d'écoute, et reconstruit la chaîne de réception, avec filtrage MAC. Si au cours de cette phase, une demande en émission est faite par l'application, celle-ci est en attente et sera effective lors de la fin du processus de réception.

En émission, le module FSK s'impose et empêche donc toute réception de message éventuel qui sera donc perdu.

Discussion sur les conflits :

Hypothèse : tous les modules sont à portée les uns des autres :

A communique vers B. C reçoit aussi le message. Ainsi, B comme C ne peuvent pas émettre puisqu'ils sont en phase de réception (B comme C ne savent pas si le message leur est destiné, ils ont juste repéré le « ##### » et se sont mis en état de réception).

Dans ce cas, le premier qui parle, oblige les autres à se taire.

Si par un gros hasard, A et C démarrent en même temps, rien ne peut les arrêter. Le checksum sera faux et les deux émissions seront avortées.

1.2. Reconstruction d'une chaîne de caractères en réception

Le principe est basé sur la donnée de la longueur de la chaîne à recevoir. Ainsi, après le préambule, le premier octet reçu correspond à la longueur, *Len*, (donc maximum théorique de 255).

Le nombre maximal d'octets que l'on peut recevoir est fixé par un #define que l'on trouve dans le fichier header *FSK_Stack.h,* :

```
#define StringLenMax 30
```

Cette limitation est modifiable (255 maximum !). Elle est d'une longueur raisonnable car les tableaux de réception sont définis à l'avance et non détruits.

Lors de la réception, un test sur la longueur est effectué. Elle ne doit être ni trop courte, ni trop longue (voir la partie traitant des erreurs).

1.3. Définitions des diverses trames émises et reçues

1.3.1. Réception

La trame reçue au niveau UART (sortie du module FSK RT606) est composées d'octets encapsulés par un bit de Start et 1 bit de Stop. La vitesse maximale est de 38400 Bds.

Cette trame est appelées ***PhyFrame*** : |#####|Len|Data|CheckSum|. Len est le nombre d'octets reçus, préambule exclu.

La trame est récupérée dans le tableau ***IncomingMssg[Len-1]*** qui contient |Data|CheckSum| si la longueur *Len* n'est ni trop courte ni trop longue, sinon rien n'est récupéré. Le champ Len est enlevé donc la longueur de la table est Len-1.

Ensuite, un checksum est opéré. Si celui-ci est OK, le message est recopié dans un autre tableau dont on enlève le champ checksum.

Ce tableau s'appelle ***StrReceived[Len-2]*** . Len-2, puisqu'il n y a plus le byte checksum. Il ne contient plus que le champ |Data|. Cette nouvelle trame, |Data|, est appelée **MACFrame :** |Data| = |@Src|@Dest|Data'| , où *Data'* est la suite de données utiles.

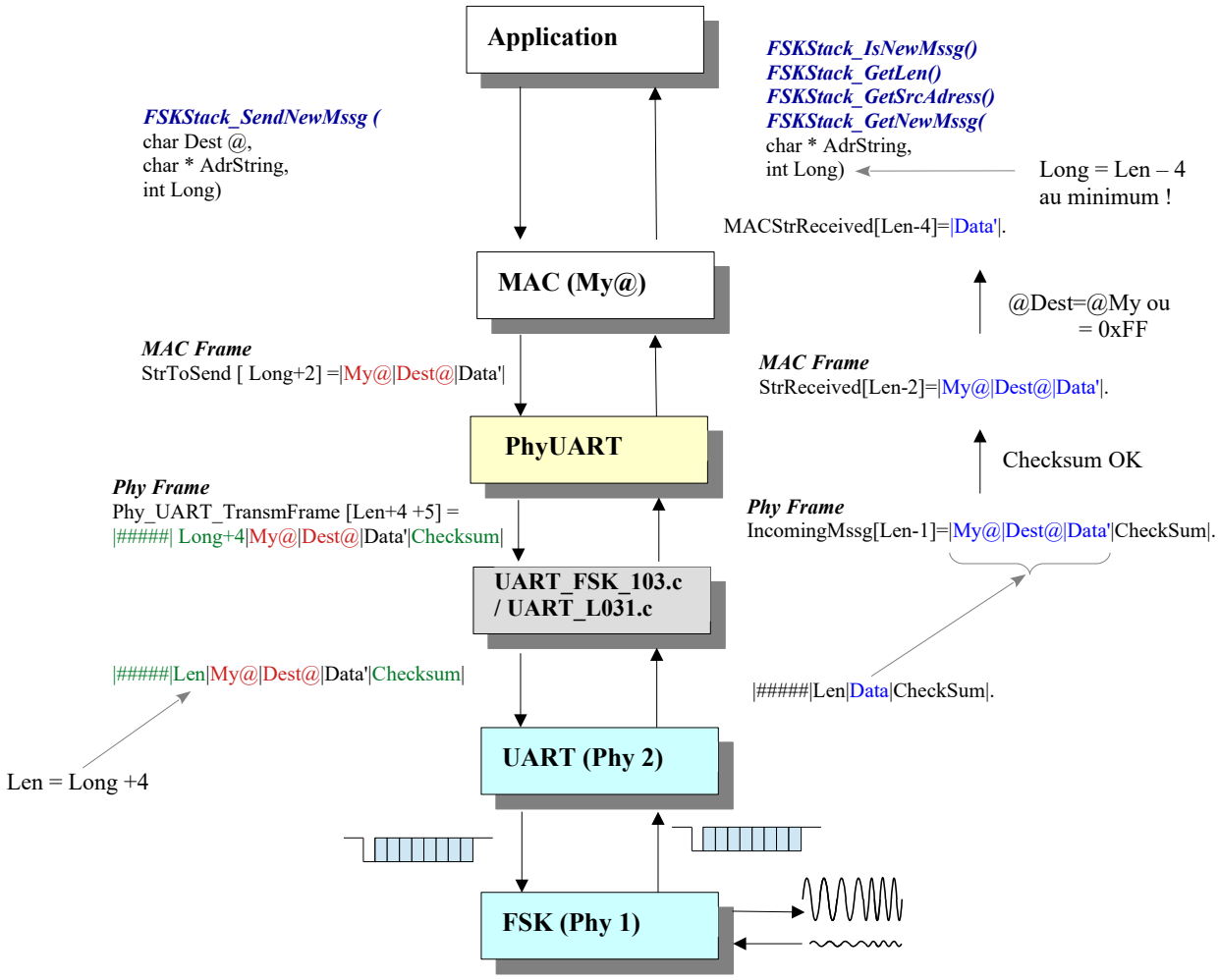
Ensuite, cette trame réduite est envoyée à la couche MAC. Le processus teste alors si l'adresse @Dest correspond à @My ou bien s'il s'agit d'un broadcast. Si c'est le cas, une nouvelle table est remplie :

le tableau ***MACStrReceived[Len-4]*** . La table ne contient plus que |Data'| , elle a donc une longueur diminuée de 2 par rapport à la table précédente, donc de 4 par rapport à la longueur d'origine.

1.3.2. Emission

Pour l'émission, le processus inverse est opéré. L'utilisateur précisera le champ de donnée |Data'|, sa longueur, ainsi que l'adresse de destination. Le passage dans chacune des couches vient construire petit à petit la trame physique envoyée en FSK. Le diagramme suivant illustre l'émission et la réception.

1.3.3. Schématisation des trames mises en œuvre



2. La machine à états de la pile

La pile est gérée par une machine à états synchrone.

2.1. Les variables associées aux transitions :

Transitions pour quitter l'état ***Init***

PhyUART_Start

type char,

Set : fct PhyUART_StartFSM*(),*

Reset : jamais

Transitions pour quitter l'état ***WaitForHeader***

NewStrToSend

type char,

Set : fct *PhyUART_SendNewMssg(),*

Reset : dès qu'il a été pris en compte (entrée état *Framing*).

Message envoyé

type ce n'est pas une variable. C'est la fin de transmission UART vers FSK qui se fait en polling. Cela correspond donc à la fin de la fonction print de l'UART,

Set : Lors de la fin de transmission effective (flag de l'UART),

Reset : automatique ou à faire logiciellement dans la fonction print UART.

UART_HeaderDetected

type char,

Set : dans le callback UART, si la séquence de ##### est trouvée,

Reset : dès qu'il a été pris en compte (entrée état *ReadingFrame*).

Transitions pour quitter les états ***ReadingFrame, Checksum, UpdateMssgForMAC***

Time Out

type char,

Set : fct *PhyUART_GetTimeOut_Status(),*la variable. *PhyUART_Mssg.Error* est alors mise à *TimeOutError*.

Reset : dès qu'un autre chronométrage est lancé. *PhyUART_Mssg.Error* est remis à *NoError* au début de l'état *ReadingFrame*. L'utilisateur peut donc lire l'erreur uniquement dans une fenêtre temporelle qui se

referme à la réception d'une nouvelle chaîne.

Len OK

type bool, test

Set : test de la longueur Len : elle doit être inférieure ou égale à la valeur maximale définie dans le .h, *StringLenMax*. Elle doit aussi être supérieure ou égale à la valeur minimale définie dans le .h, *StringLenMin*. Si ce n'est pas le cas, *PhyUART_Mssg.Error* est mise à *LenError.*

Par défaut *StringLenMax = 30 (ce qui implique une charge utile de 26 octets dans le message MAC),*

Par défaut *StringLenMax = 5 (Byte Len + Byte @Src + Byte @Dest + 1 Byte data + 1 Byte CheckSum)*

Reset : *PhyUART_Mssg.Error* est remis à *NoError* au début de l'état *ReadingFrame*. L'utilisateur peut donc lire l'erreur uniquement dans une fenêtre temporelle qui se referme à la réception d'une nouvelle chaîne.

CheckSumError

type bool, test

Set : test de la valeur du Checksum. Si le CheckSum n'est pas bon, *PhyUART_Mssg.Error* est mise à *CheckSumError.*

Reset : *PhyUART_Mssg.Error* est remis à *NoError* au début de l'état *ReadingFrame*. L'utilisateur peut donc lire l'erreur uniquement dans une fenêtre temporelle qui se referme à la réception d'une nouvelle chaîne.

2.2. Les états de la machine

La machine est régie par les états suivants, définis par l'énumération suivante :

```
typedef enum {
        Init,
        WaitForHeader,
        ReadingFrame,
        CheckSum,
        UpdateMssgForMAC,
        Framing,
        SendMssg
}PhyUART_FSM_StateType;
```

On définit aussi des états de la pile, qui correspondent à des groupes d'états de la FSM (voir la FSM plus loin). Ces états de la pile peuvent être lus par l'utilisateur. Ce sont :

```
typedef enum {
        Ready,
        Listening,
        ReceivingMssg,
        SendingMssg,
}PhyUART_StatusType;
```

Voici le détails des états de la FSM :

2.2.1. Etat Init

Etat d'attente démarrage de la pile. Procède à toutes les initialisations des variables. L'échantillonnage synchrone de la FSM est fixé à 1ms.

2.2.2. Etat WaitForHeader

Attente du préambule #####. S'il est détecté la FSM est accélérée à 100µs pour pouvoir construire la chaîne de caractères correctement, même à 38400 Bd (un byte → 260µs). Cet état attend aussi une émission de message via l'utilisateur.

2.2.3. Etat ReadingFrame

Extrait la longueur *Len* de la chaîne, puis stocke l'ensemble dans la variable *IncomingMssg*. Si tout s'est bien passé, la FSM évolue vers l'état *CheckSum*.

2.2.4. Etat CheckSum

Vérifie la valeur du *CheckSum*, puis le cas échéant, évolue vers l'état *UpdateMssgForMAC*.

2.2.5. Etat UpdateMssgForMAC

Retire le byte *Len* et le byte *Checksum*, et stocke la nouvelle chaîne dans la variable *StrReceived*. Si l'adresse destinataire est égale à *My* ou égale à 255 (Broadcast), le message est passé à la variable *MACStrReceived* à laquelle les adresses origine et destinataire sont ôtées. La longueur est mémorisée ainsi que l'adresse d'origine pour pouvoir répondre.

NB : au début de cet état, on vérifie si ces deux chaînes de caractères ont été lues. Si ce n'est pas le cas, la variable *PhyUART_Mssg.Error* est chargée à la valeur *OverRunError* (pour *StrReceived*) et *MACOverRunError* (pour MAC*StrReceived*)

2.2.6. Etat Framing

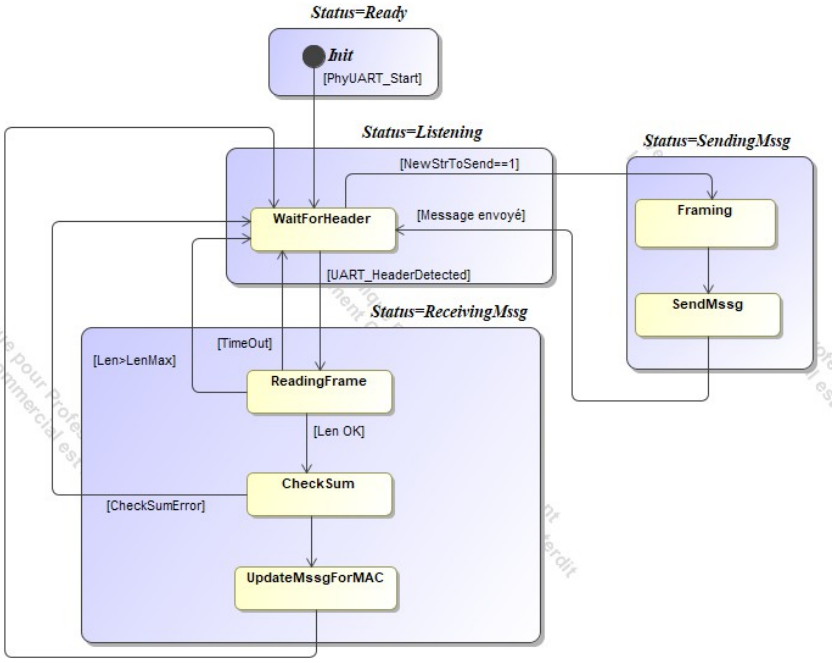
La variable chaîne de caractères *StrToSend* est encapsulée dans la chaîne *Phy_UART_TransmFrame*. (ajout du préambule, des adresses de destination, d'origine, longueur, checksum).

2.2.7. Etat SendMssg

Procède à l'envoie des octets au RT606 pour émission FSK.

NB : on envoie au début quelques caractères quelconques pour être sûr que le récepteur capte bien le préambule.

2.3. La machine à état



3. Les erreurs gérées

Le module contient une variable erreur de type énuméré. Voici l'énumération :

```
typedef enum {
		NoError,
		CheckSumError,
		LenError,
		TimeOutError,
		OverRunError,
		MACOverRunError
}PhyUART_ErrorType;
```

Toutes ces erreurs ont été présentées dans le paragraphe précédent. Précisons que celle-ci peuvent être lue uniquement dans une fenêtre temporelle entre la fin du processus de réception en cours et la prochaine réception.

4. L'interface utilisateur

4.1. La variable (attribut) PhyUART_Mssg

Le module contient une grosse variable structurée qui regroupe bon nombre des variables vues précédemment. Elles sont parfois accessible via des getter et setter que nous verrons par la suite. La variable s'appelle ***PhyUART_Mssg*** elle est de type ***PhyUART_Mssg_type*** :

```
struct PhyUART_Mssg_type
{
        char StrReceived[StringLenMax-2];  // |Org@|Dest@|Data| , il n'y a plus Len ni Checksum : inaccessible
        char LenStrReceived;  // longueur de la chaîne StrReceived : accessible R
        char NewStrReceived; // indicateur nouvelle chaîne arrivée du réseau ##### : accessible R
        // ---< Ajout MAC >-----//
        char MACMatch;  // indicateur validité MAC , @dest =My ou @dest = 0xFF : inaccessible
        char My; // Adresse MAC propre du module : accessible W
        char MACStrReceived[StringLenMax-4];    // |Data| : on enlève les adresses d'origines et de destination :  accessible R
        char MACLenStrReceived; // longueur de la chaîne StrReceived : accessible R
        char MACNewStrReceived; // indicateur nouvelle chaîne MAC arrivée du réseau ##### : accessible R
        char MACSrcAdress;  // Adresse MAC origine du message  : accessible R
        // ---< Fin Ajout MAC >-----//
        char StrToSend[StringLenMax]; // données brutes à transmettre de puis l'application : accessible W
        char LenStrToSend; // longueur de la chaîne à transmettre, cad nombre de données précédente : accessible W
        char NewStrToSend; // indicateur nouvelle chaîne à émettre: inaccessible
        PhyUART_StatusType Status; // état de la pile (voir énumération $2.2) : accessible R
        PhyUART_ErrorType Error; // erreur de réception : : accessible R
}PhyUART_Mssg;
```

4.2. Les fonctions API de la pile

4.2.1. Réglages header, User Define

Temps réel, interruption externe (CD), UART, Ck FSM

#define UART_Prio_CD 0 // priorité donnée à l'interruption externe connectée à CD. Elle est à la valeur prioritaire, donc 0.

#define UART_Prio (UART_Prio_CD+1) // priorité donnée à l'UART en réception d'octet. Niveau juste au dessus de CD.

#define PhyUART_FSM_Prio (UART_Prio+1) // priorité donnée au timer qui cadence la FSM. Juste au dessus de celle de l'UART.

NB : Le user n'a qu'à régler le premier #define. Il est recommandé

#define TIM_PhyUART_FSM TIM2 // Précise le timer associé au cadencement de la FSM.

#define PhyUART_BdRate 38400 // précise le débit de l'UART pour la FSK

NB : L'UART utilisée est définie dans le module immédiatement inférieur concernant la FSK (voir architecture logicielle)

#define StringLenMax 30 // Nbre d'octets maximum reçus par l'UART (hors préambule) : |Len | data (dont @) |Checksum| ,

#define StringLenMin 5 // Nbre d'octets minimum attendu (au moins un byte utile)

4.2.2. Fonctions générales (3)

void FSKStack_Init(char My) :

- fixe l'@My,
- lance la configuration de l'UART retenue,
- place le module FSK RT606 en réception,
- Lance la configuration du Timer FSM à 1ms,
- lance le systick pour la gestion des TimeOut,
- détermine le TimeOut, en ms (constante *PhyUART_TimeOut*) pour une chaîne maximale autorisée + 10% calculé en fonction des #define de l'utilisateur (*StringLenMax* et *PhyUART_BdRate*).
- Démarrage de la FSM (voir ci après)

PhyUART_StartFSM (privé hors API)

- Macro qui démarre la pile et qui la place dans l'état d'attente WaitForHeader (Listening)

char FSKStack_IsNewMssg(void);

- renvoie l'indicateur d'arrivé *MACNewStrReceived,* donc 1 si une nouvelle chaîne est arrivée.

4.2.3. Getter (5), setter (1)

Ces fonctions sont directement liées à la variable *PhyUART_Mssg* présenté plus haut.

char FSKStack_GetLen(void);

- renvoie la longueur effective de *MACStrReceived.*

*int FSKStack_GetNewMssg (char * AdrString, int Len);*

- permet de lire le tableau *MACStrReceived* et le recopie à partir de *AdrString* avec un nombre de caractères égal au paramètre *Len*.
- Si la longueur Len est plus petite que la longueur effective de *MACStrReceived*, la fonction renvoie -1. Sinon elle renvoie 0.

char FSKStack_GetSrcAdress(void);

- renvoie l'adresse source du message *MACStrReceived.*

PhyUART_StatusType FSKStack_Get_Status(void);

- renvoie l'état dans lequel se trouve la pile, parmi les 4 « macroétat » possibles (voir § *les états de la machine*).

  NB : la fonction réelle est *PhyUART_Get_Status* rendue accessible par un *#define*.

PhyUART_ErrorType FSKStack_Get_Error(void);

- renvoie le statut d'erreur de la pile (voir § *les erreurs gérées*).

  NB : la fonction réelle est *PhyUART_Get_Error* rendue accessible par un *#define*.

*int FSKStack_SendNewMssg (char DestAdr, char * AdrString, int Len);*

- Envoie dès que possible la chaîne de caractère d'adresse *AdrString* au destinataire dont l'@ est *DestAdr*. Le nombre d'octets à envoyé (data pure hors tout) est *Len*.
- Si la longueur *Len* est supérieure au maximum autorisé (*StringLenMax*-4), la fonction de fait rien et renvoie -1. Si tout se passe bien, elle renvoie 1.

5. Détails de mise en œuvre

5.1. La gestion du temps

La FSM, cœur de la pile, repose sur l'utilisation de deux timers pour assurer les fonctions de :

- cadencement synchrone de la FSM
- gestion des timeout

5.1.1. Le cadencement synchrone de la FSM

Le timer utilisé peut être quelconque. Il nécessite juste d'être capable de générer une interruption. Le callback associé, écrit dans le module s'appelle *PhyUART_FSM_Progress()*. Il s'agit tout simplement de la FSM. Le timer en question est défini à travers un #define dans le .h associé : *#define TIM_PhyUART_FSM TIM2.* Comme nous l'avons vu, cette durée sera réglée soit à 1ms pour soulager le programme en background, soit à 100µs pour pouvoir échantillonner chaque octet en provenance du RT606, à 38400 Bd.

5.1.2. La gestion du TimeOut

Ici c'est obligatoirement le Systick qui est utilisé.

NB: Cela implique que si du code automatique est généré (HALL, LL), il faut neutraliser tout ce qui implique le Systick !

Le principe est le suivant :

- le systick est réglé à 100µs, en interruption,
- un compteur interne est simplement incrémenté, il indique donc le nombre de fois 100µs depuis le démarrage,
- un getter permet de récupérer la valeur de ce compteur et donc permet à la pile de savoir où on en est depuis le lancement du Systick.

NB : il s'agit d'une variable 32 bits. La durée maximale de comptage est donc (2^32-1)*100µs = 119 heures.

Ces fonctionnalités de bas niveau sont gérée entièrement dans le module *Timer_F103.c/.h*.

La gestion effective des TimeOut se fait dans le module *TimeManagement.C/.h.*

La fonction *void TimeManag_TimeOutStart(TimeBaseName Name , int ms)* règle une variable interne (une date) correspondant à la valeur actuelle du compteur Systick à laquelle on ajoute la durée demandée en ms (variable *ms*).

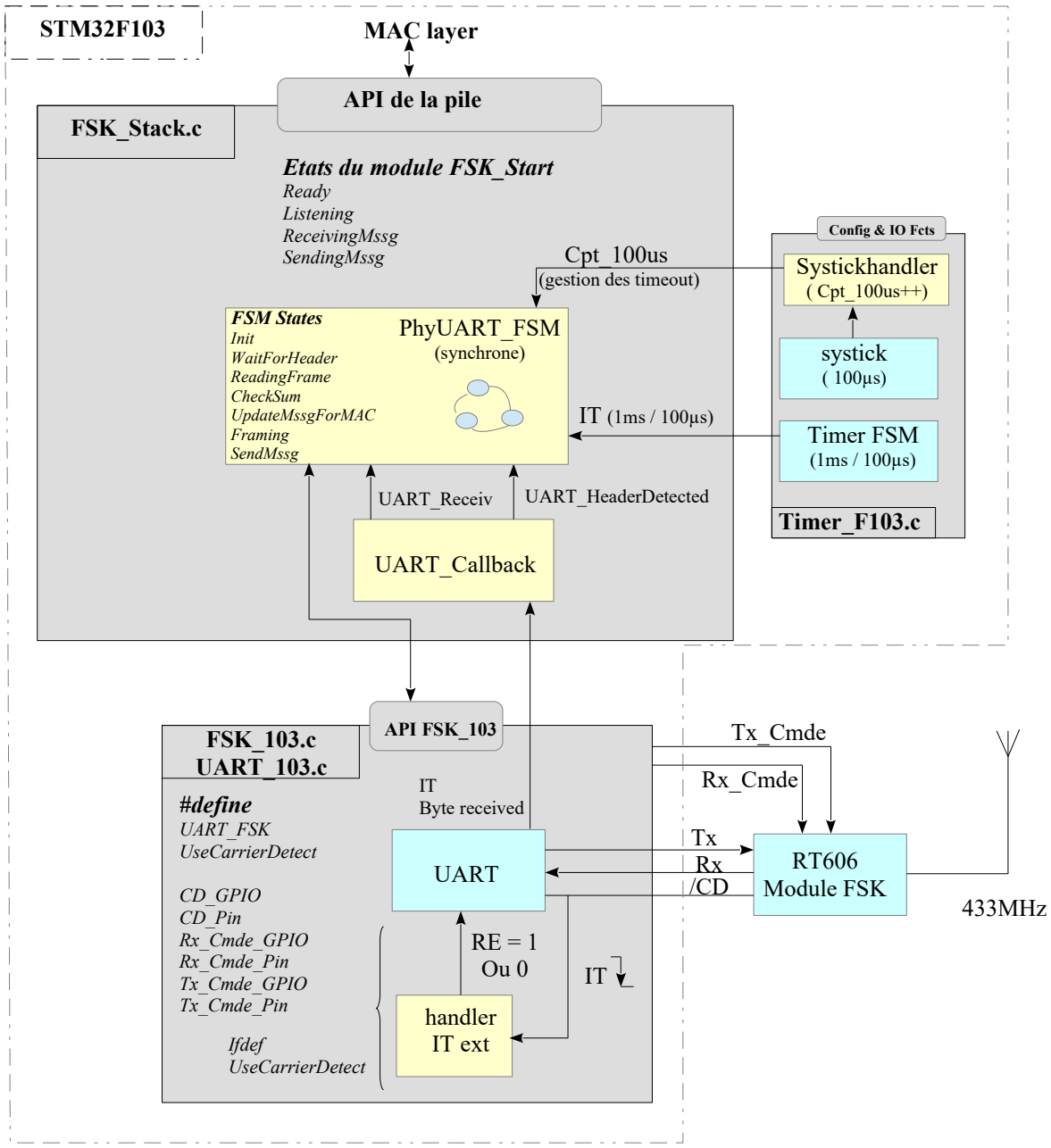
Une seconde fonction, *char TimeManag_GetTimeOutStatus(TimeBaseName Name)* retourne 1 si la date précédemment définie est dépassée. Cette fonction permet donc de savoir si le temps est écoulé ou pas.

NB : supposons que la valeur actuelle de la variable soit 0xFFFF FFFF (soit -1), et que l'on ajoute 100 pour chercher un timeout à 10ms (100*0.1ms). La date calculée vaudra alors 99. Autrement dit, la calcul de date ne craint pas le débordement du compteur 100µs du Systick.

5.2. L'architecture

5.2.1. Architecture logicielle et matérielle

Le schéma ci-dessous illustre l'architecture matérielle/logicielle de la pile :



Utilisation du Carrier Detect du module FSK RT606

En réception, il se peut que le module soit inondé d'octets parasites, ce qui sollicite inutilement l'UART et peut être source d'erreur. En effet, le récepteur, ne détectant aucune porteuse (pas de message à recevoir) se met en sensibilité maximale et reçoit des parasites.

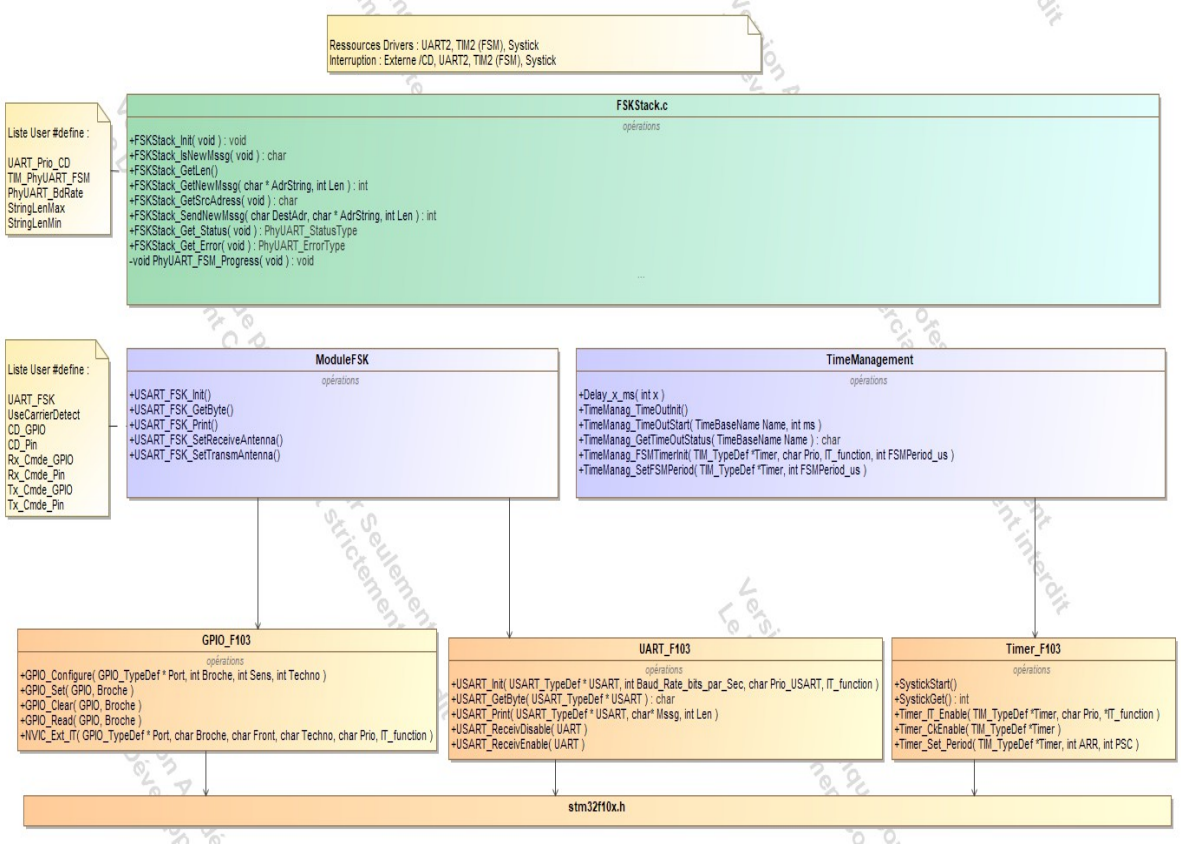
Pour éviter cela, on peut utiliser le signal /CD (not Carrier Detect) qui passe à 0 si une porteuse de puissance suffisante est détectée. Dès lors, le module UART se retrouve validé en réception et peut donc générer des

interruptions UART. Comme ce temps de réveil peut être un peu long, on envoie systématiquement avant le préambule identifiant ##### une suite de 5 octets quelconques pour que l'UART puisse se réveiller et ne pas rater un des #.

5.2.2. Architecture logicielle pour STM32F103RB

Elle se découpe en trois partie,

- le code proprement dit de la pile. Entre une implémentation sur STM32F103RB et sur un STM32L031F6P7, seuls les headers sont modifiés au niveau des inclusions (deux lignes). Le source c est inchangé,
- le code intermédiaire qui gère d'une part le module FSK et les ressources nécessaires au temps (systick et timer FSM) d'autre part. Les headers sont identiques (juste les inclusions qui changent entre les deux versions STM32F103RB et STM32L031F6P7). Le code C est quasi identique dans les deux versions. La fonctions délais est modifié puisque le cadencement en F103 est de 72MHz et seulement de 24MHz sur L031. Pour le module FSK, dans la version 103, les broches y sont entièrement configurées. Dans la version L031, la configuration est ailleurs (cadre du remote Device). Donc les lignes sont enlevées.
- La couche driver. Pour la version STM32F103RB tout est fait au registre. Pour le STM32L031F6P7 c'est aussi une conception aux registres avec aide de quelques petites lib LL.



Dans tous les cas, l'utilisateur doit opérer tous les réglages #define décrits dans le diagramme de classes afin de bien paramétrer la pile. Ceux ci sont indiqués en commentaire sur les diagrammes de classes.

5.2.3. Architecture logicielle pour STM32L031F6P7