Gestion de temps dans le système
Solar Management System for air conditioners

Table des matières

1. Overview

Le temps intervient à plusieurs niveaux dans l'application. Les *remote Devices* doivent envoyer des requêtes vers la *Smart Gateway* à des dates précises. Il est donc nécessaire de disposer d'un système d'**horodatage** (**Stamp**). Par ailleurs, une dérive de l'horloge interne est possible. Il faut donc pouvoir **compenser les échéances théoriques** logiciellement (au niveau SGw) pour que les transactions puissent avoir lieu à des dates précises.
Enfin, le temps intervient aussi au niveau des ***timeout***, pour éviter toute fonction bloquante.

La gestion du temps dans l'application se fait à quatre niveaux :

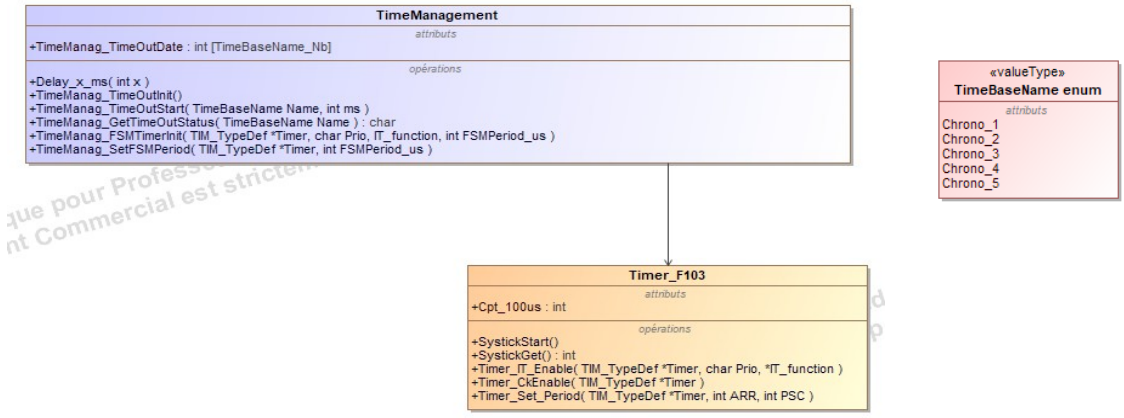
- la gestion des ***timeout généraux***. Le timer utilisé est le *Systic*k, configuré en boucle à 100µs. Le module est *TimeMamagement.c/.h.*

- la gestion du ***timeout*** **spécifique de la pile *UARTStack***. Le timer utilisé possède l'alias *TIM_UARTStack* configuré par défaut en boucle à 100ms. Cette gestion très simple se fait directement avec les fonction de la lib de bas niveau Timer. Le module *UARTStack.h/.c* gère donc ce *timeout*.

- la gestion de l'***horodatage*** (mesures au niveau du système de toutes les dates d'arrivées des diverses transactions avec les *RmDv*, synchronisation avec la HMI). Le timer utilisé possède l'alias *TIMER_TimeStamp* configuré en boucle à 1 seconde. C'est le module *TimeStampManagement.h/.c* qui gère l'horodatage,

- la gestion de la ***compensation des délais*** pour assurer des transactions RmDv / SGw à des dates précises.

2. Gestion des timeout généraux



2.1. Détails, explications

La fonction d'initialisation démarre un compteur (une variable *int Cpt_100us*) qui s'incrémente grâce au *Systick* toutes les 100µs. Ce compteur constitue donc la référence de temps absolu pour la gestion de plusieurs *timeout* (5 maximum, *Chrono_1* à *Chrono_5*).

Le tableau *int TimeManag_TimeOutDate[]* contient les 5 entiers 32 bits qui gère chacun des 5 chronomètres.

Par exemple, l'appel de la fonction *TimeManag_TimeOutStart(Chrono_2, 50)* démarre le chronomètre n°2 ce qui veut simplement dire que la variable *TimeManag_TimeOutDate[Chrono_2]* contient alors la date d'échéance du chronomètre calculée sur la base du temps spécifié (ici 50ms) et de la valeur du compteur de référence.

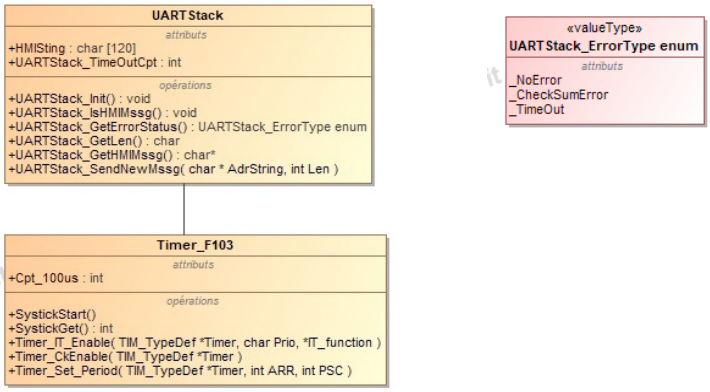
La fonction *TimeManag_GetTimeOutStatus(...)* renvoie 1 ou 0 selon que le temps spécifié est atteint ou pas (par simple comparaison entre la date d'échéance et le compteur de référence).

NB : le calcul ne craint pas le débordement du compteur de référence.
NB : 3 chronomètres sont utilisés. Les
NB : ce module gère aussi le cadencement de la FSM (Finite State Machine) de la pile. Pour plus d'information voir *FSKStack_UG_Light.pdf*, §5.1.1

2.2. Exemple d'utilisation

- *TimeManag_TimeOutInit();* // lancement du systick, avec incrémentation sur interruption de la variable *int Cpt_100us* (dans le module *Timer_F103.c/.h*),
- *TimeManag_TimeOutStart(Chrono_2, 50) ;* // Calcule et mémorise la date d'échéance du chronomètre n°2 dans la case du tableau *TimeManag_TimeOutDate[Chrono_2],*
- *if (TimeManag_GetTimeOutStatus(Chrono_FSKStack)==1)* // Teste si le timeout est effectif ou non.

3. Gestion du timeout spécifique à la pile UARTStack



3.1. Détails, explications

La pile *UARTStack* gère la communication entre la *Smart Gateway* et la *HMI*. Les *timeout* se chiffrent en centaines de ms. Il a été décidé d'exploiter un timer spécifique pour cela (*TIM_UARTStack)*. Le timeout est donc géré uniquement par les fonctions du module *Timer_F103.c/.h.* Là encore on utilise une variable compteur, *UARTStack_TimeOutCpt*.

Dès le début d'une transaction, marquée par la réception asynchrone du premier byte en provenance de la *HMI*, le timer spécifique, *TIM_UARTStack,* est lancé. Le *timeout* est systématiquement attendu (sous interruption) pour mettre à jour le statut d'erreur.

Le quantum de temps est relativement large, 100ms.

3.2. Exemple d'utilisation

Préparation :

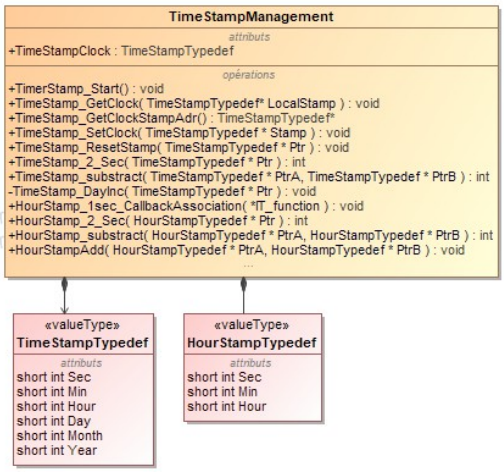
- *Timer_CkEnable(TIM_UARTStack);* // validation du périphérique
- *Timer_Set_Period(TIM_UARTStack, 10000-1, 720-1);* // ajustement du quantum à 100ms
- *Timer_IT_Enable(TIM_UARTStack,1, UARTStack_TimeOut);* // association de la fonction d'interruption à échéance des 100ms

Fonction d'interruption UARTStack_TimeOut() :

- *UARTStack_TimeOutCpt++ ;* // incrémentation du compteur
- if (UARTStack_TimeOutCpt==TimeOut_x100ms) // test si *timeout* atteint. Dans l'exemple, *TimeOut_x100ms vaut* 5, la valeur du *timeout* ici est donc de 500ms.

4. L'horodatage

4.1. Détails, explications



L'horodatage est géré au niveau de la *Smart Gateway*. Il est important dans l'application, il permet de gérer le temps absolu à tous les niveaux y compris l'IHM

Deux ensembles sont au cœur de l'horodatage :

- un timer, *TIMER_TimeStamp* qui est associé à une fonction d'interruption (privée dans le module *TimeStampManagement*) réglée à 1 seconde. La fonction d'interruption incrémente de 1 seconde la variable *TimeStampClock* (donc tous les champs en cascade),
- une variable *TimeStampClock*, de type *TimeStampTypedef* qui détient l'heure absolue du système.

La structure *TimeStampTypedef* est définie comme suit :

```
typedef struct
{
        short int Sec;          /* 16 bits*/
        short int Min;          /* 16 bits*/
        short int Hour;         /* 16 bits*/
        short int Day;          /* 16 bits*/
        short int Month;        /* 16 bits*/
        short int Year;         /* 16 bits*/
}TimeStampTypedef;              /* Total 12 bytes*/
```

La variable, par défaut est initialisée à la date du *1 Jan 2024 00:00:00*.

Un second type de structure est utilisé, réduite aux heures, minutes et seconde :

```
typedef struct
{
        short int Sec;          /* 16 bits*/
        short int Min;          /* 16 bits*/
        short int Hour;         /* 16 bits*/
}HourStampTypedef;              /* Total 6 bytes*/
```

Elle est utilisée dans les transactions RmDv – SGw pour calculer plus simplement les prochaines dates de transaction.

4.2. Détails des fonctions importantes du module

*4.2.1. void TimeStamp_SetClock(TimeStampTypedef * Stamp)*

La fonction cale l'horodatage absolu à la valeur d'horodatage passée en paramètre : le système est mis à l'heure.

4.2.2. void TimerStamp_Start(void)

Initialise le timer *TIMER_TimeStamp* à une seconde de débordement, l'interruption timer est activée, le timer est lancé.

4.2.3. void TimeStamp_GetClock(TimeStampTypedef * LocalStamp)

La fonction charge la variable pointée par l'adresse passée en paramètre avec l'horodatage courant. On récupère ainsi l'horodatage ablsolu.

*4.2.4. TimeStampTypedef * TimeStamp_GetClockStampAdr(void)*

La fonction renvoie l'adresse de l'horodatage absolu.

*4.2.5. void TimeStamp_ResetStamp(TimeStampTypedef * Ptr);*

Met à 0 l'horodatage dont l'adresse est spécifiée en paramètre.

*4.2.6. int TimeStamp_2_Sec(TimeStampTypedef * Ptr)*

Convertit un horodatage en seconde. Pour cela on utilise la référence absolue du 1 Jan 2024, 00:00:00.

*4.2.7. int TimeStamp_substract(TimeStampTypedef * PtrA,TimeStampTypedef * PtrB)*

Renvoie la différence en seconde Horodatage A – Horodatage B. L'opération nécessite d'abord une conversion en seconde, puis la différence est faite. La fonction peut donc renvoyer autant une différence positive que négative.

*4.2.8. void TimeStamp_DayInc(TimeStampTypedef * Ptr)*

Incrémente de un jour l'horodatage passé par adresse.

*4.2.9. HourStamp_1sec_CallbackAssociation(void (*IT_function) (void))*

Cette fonction permet d'ajouter un appel de fonction à lors de l'interruption timer toutes les secondes. Le nom de la fonction est passé en paramètre.

*4.2.10. int HourStamp_2_Sec(HourStampTypedef * Ptr);*

A l'instar de la fonction *TimeStamp_2_Sec*, la fonction convertit en seconde un horaire. La référence est logiquement *00h:00mn:00s*.

*4.2.11. HourStamp_substract(HourStampTypedef * PtrA, HourStampTypedef * PtrB)*

Identique à TimeStamp_substract mais réduite à des horaires. Le résultat est en secondes. Nombre de secondes = sec A- sec B.

*4.2.12. void HourStampAdd(HourStampTypedef * PtrA, HourStampTypedef * PtrB)*

Ajoute deux horaires A = A+B. Contrairement à la fonction précédente, *HourStamp_substract*, le résultat est un horaire.

4.3. Exemple d'utilisation

Quelque soit l'exemple d'application (voir chapitre 5 pour un exemple entre RmDv et SmGw), il est indispensable de :

- mettre à l'heure l'horodatage absolu : TimeStamp_SetClock(...);
- Démarrer le timer associé : TimerStamp_Start

5. La compensation des délais entre RmDv et SGw

5.1. Présentation

Le *RmDv* possède peu de précision temporelle dû au fait que l'on utilise le circuit RC interne pour cadencer le *WUTimer*. Par contre, côté *SGw*, la précision temporelle est très bonne puisque c'est le quartz de 8MHz externe qui est à la base du cadencement du processeur. C'est donc la SGw qui va « recaler » les horloges des *RmDv*, ou plus précisément, la *SGw* va opérer un ajustement (dilatation ou compression du temps théorique) pour atteindre les délais effectifs. Ceux ci sont de 30mn en journée, et de 6 à 8h pour la nuit.

5.2. Détails, explications

Les explications qui suivent sont mises en œuvre dans le module *DataFromRmDv* qui a bien d'autres rôles. La fonction qui concerne le temps est

int RmDvData_GenerateNextTimeInterval(RmDvDataTypedef RmDvData)*

La fonction passe par adresse la variable *RmDvData*. Elle concerne un *RmDv* particulier caractérisé par son *ID* (voir détail de la structure ci-dessus). La structure incorpore une sous-structure *Delay_Typedef* dont les champs gèrent directement l'horodatage et les prochains rendez-vous de transaction.

L'objectif de cette fonction est de produire l'intervalle de temps qui sépare la date d'arrivée de la présente transaction (*now*) de la date de la prochaine transaction. Cette valeur est exprimée en seconde (*int)*.

La fonction est capable de générer la date du prochain rendez-vous (par exemple un *stamp* à 9h07 donne une date cible à 9h30) et donc d'en déduire cet intervalle de temps (23mn dans l'exemple). Le problème qui se pose est que la précision des *RmDv* est mauvaise, et donc, la *SGw* doit pouvoir évaluer le *facteur de dilatation temporelle*, qui évalué par rapport à la transaction précédente :

$$Facteur\ de\ dilatation\ temporelle = \frac{mesure\ intervalle\ précédent}{intervalle\ précédemment\ transmis\ au\ RmDv}$$

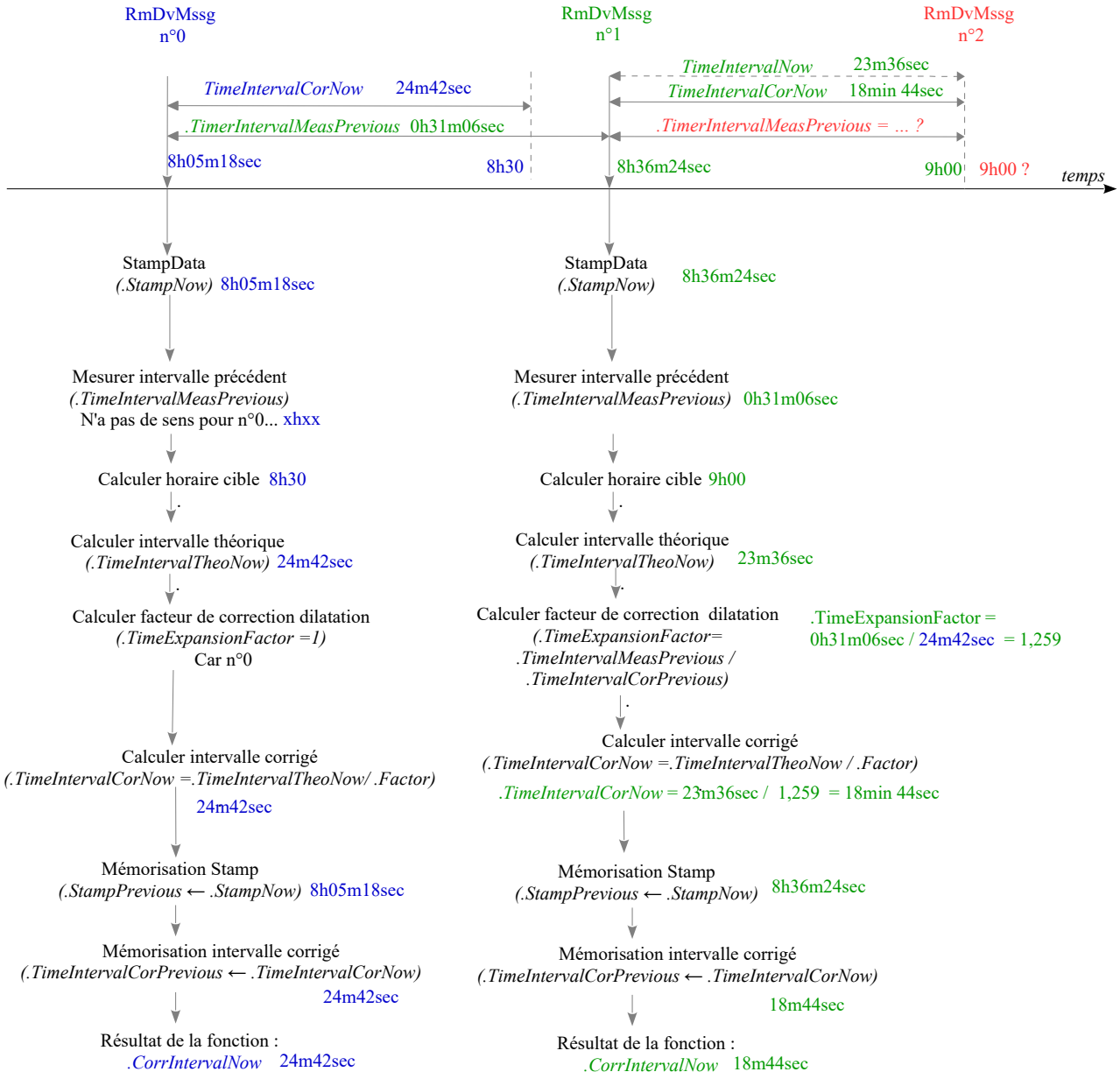
Si le facteur est plus petit que 1, c'est que le temps file trop vite au niveau du *RmDv*. Inversement si le facteur est supérieur à 1, l'horloge du *RmDv* est trop lente.

La *SGw* va donc transmettre un temps **corrigé :**

$$Intervalle\ transmis\ corrigé = \frac{intervalle\ de\ temps\ calculé}{Facteur\ de\ dilatation\ temporelle}$$

5.3. Algorithmes

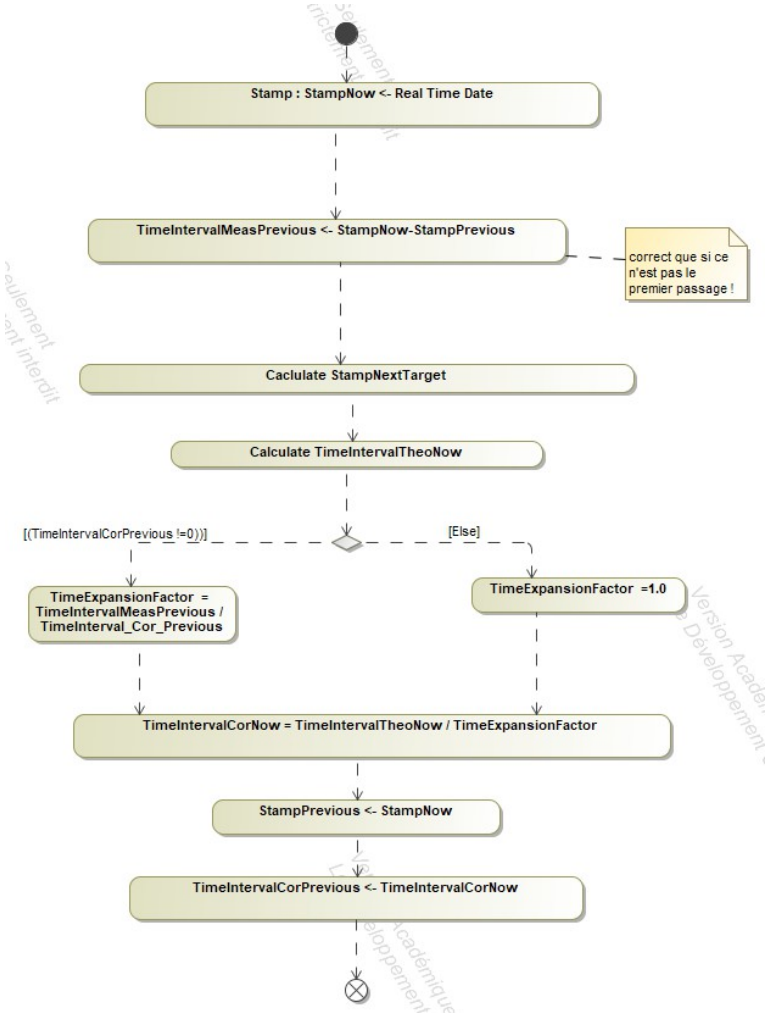
La figure ci-dessous montre un chronogramme avec 3 transactions initiées par un *RmDv*. On y voit aussi la séquence d'opérations permettant d'obtenir les prochains intervalles. Le nom de chaque variable est celui du champ associé dans la structure *Delay_Typedef* .



NB : La transaction n°0 ne peut pas être corrigée puisque la correction se fait se les bases d'informations de la transaction précédente. Le facteur de correction est donc verrouillé à 1.0,

NB : dans la mise en œuvre, le facteur de correction est limité à 0.8 et 1.2. En effet, la précision des *RmDv* est en dessous de +/-20%.

Voici l'algorithme implémenté dans la SmGw pour la fonction *RmDvData_GenerateNextTimeInterval :*



6. Gestion de la date absolue

La gestion menée par la *SGw* ne peut pas démarrer tant que le système n'est pas mis à l'heure. Pour se faire, il faut qu'il y ait eu une première transmission de données de l'IHM, sans quoi le système répond « Stop » aux *RmDv* et demande une nouvelle transmission après 5mn.

Pour cela, la variable globale *ClockUpdated* est utilisée. Elle vaut 0 tant qu'aucune donnée n'est reçue depuis l'*IHM*, 1 sinon.

A chaque transmission de l'*IHM*, l'heure est comparée avec celle du *SGw*. Si elle dépasse 5 secondes, un mise à l'heure de la *SGw* est faite.