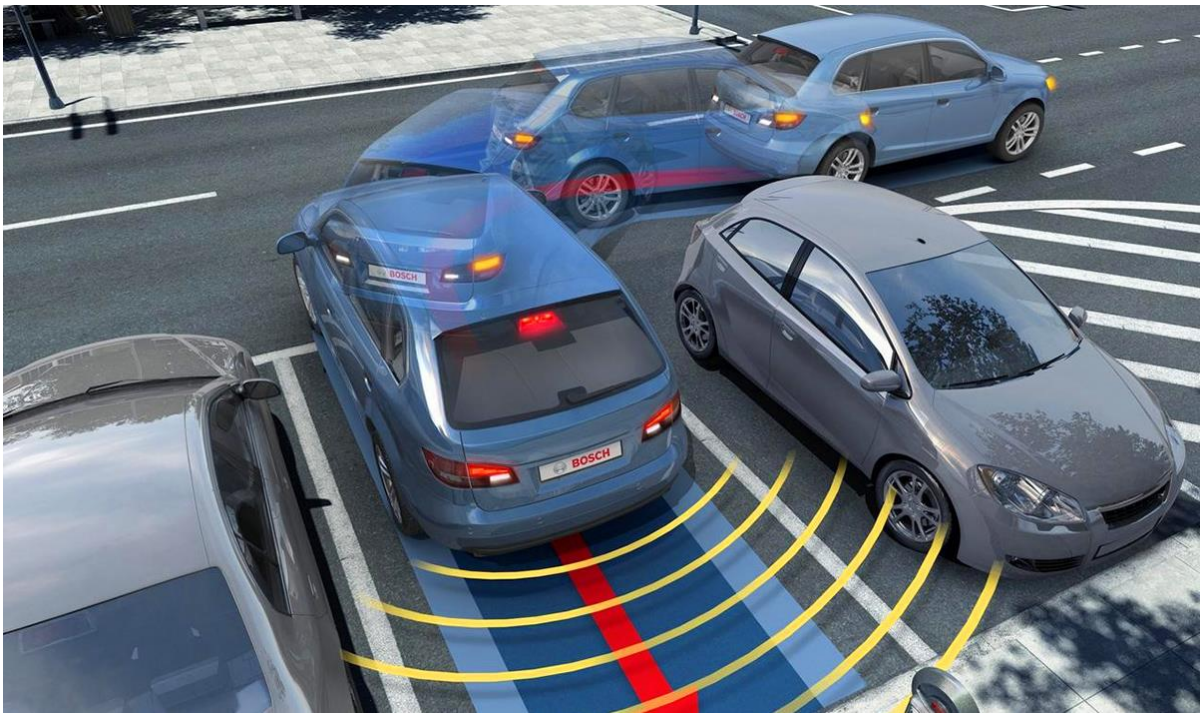
Rapport de Projet GSE 5

2015 – 2016

Réalisation d'un radar de recul



Alexandre Hochart

Mickaël Renault

Sommaire

Introduction

Nous allons mettre en perspective les différents domaines d'étude qui composent notre cursus en Génie des Systèmes Embarqués par l'intermédiaire d'un projet à la fois riche et concret, découpé en plusieurs étapes de développement. Ce projet est proposé et encadré par M. Dekneuvel, enseignant-chercheur au sein de l'Ecole Polytechnique de l'Université de Nice, et s'étale sur toute la durée du semestre 9. Le système à concevoir est un radar de recul à capteur ultrason, dont le détail est précisé dans le cahier des charges de la partie suivante.

Nous développerons dans ce rapport les phases de spécification du système, de validation du modèle fonctionnel, de modélisation du système de communication interne, des interfaces, et enfin de prototypage.

Chaque étape donne lieu à un rapport intermédiaire témoignant du travail réalisé, des choix opérés et leur justification. Ces conditions, de même que l'environnement de travail, sont pour nous l'occasion de renforcer nos qualités organisationnelles, d'adaptation, de gestion du temps, de travail en équipe, de constance et d'assiduité. La dimension de ce projet nous donne donc les moyens de travailler tout autant l'électronique technique, que la gestion de projet.

Rappel du cahier des charges

Dans ce projet, on se propose de décrire et de valider la solution architecturale d'un système de radar de recul tel qu'il en existe dans la plupart des véhicules de nos jours pour l'assistance au stationnement (cf figure 1).

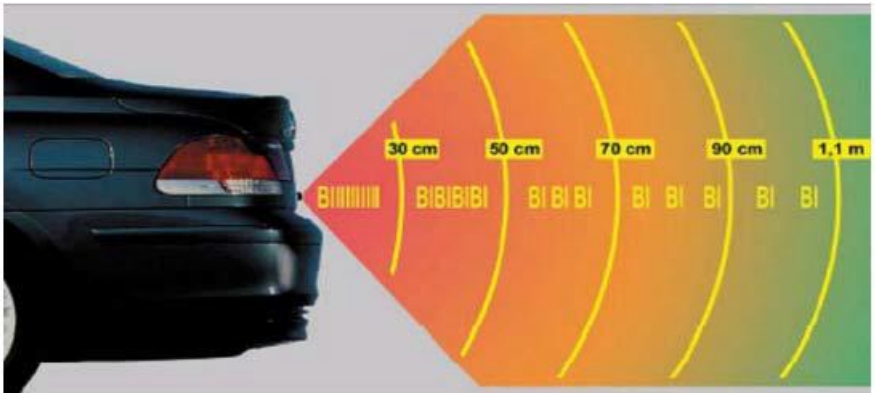


Figure 1. Assistance au stationnement à l'aide de radar à ultrasons

Typiquement, un tel dispositif se compose de 2 parties :

- Le radar : situé à l'arrière du véhicule, il peut être multiple (généralement au nombre de 4) mais il sera considéré comme unique dans le cadre du projet.
- Les interfaces : situées à l'avant du véhicule (au niveau du poste de conduite), elles seront dans le cadre du projet au nombre de 2 :
    1. Une interface d'affichage de la valeur de la mesure sur écran de type LCD.
    2. Une interface sonore de type mini sirène dont le nombre de bips sonore sera ajusté en fonction de la proximité de l'obstacle.

La technologie utilisée pour la détection d'obstacle est celle des ultrasons dont la fréquence caractéristique de fonctionnement se situe au voisinage des 40Khz (donc dans une plage de fréquence non audible par l'oreille humaine). Le principe de la mesure de distance est celui de la mesure du temps de vol. Comme le montre la Figure 2, l'entité émetteur/récepteur US se comporte comme un système convoluant le signal en entrée (burst) par sa fonction de transfert. On peut noter ainsi des oscillations résiduelles lors de l'arrêt de l'émission.

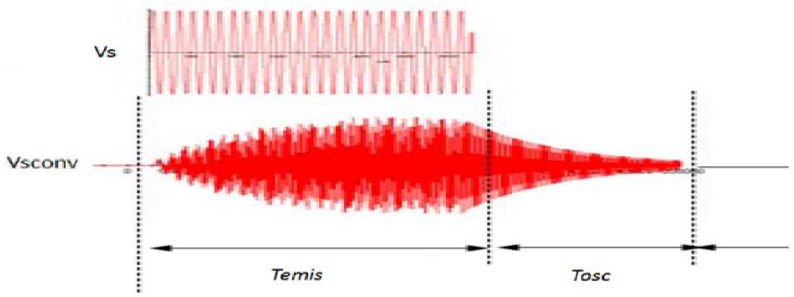


Figure 2 : Réponse impulsionnelle de l'émtteur US à un signal carré

Les figures ci-dessous montrent un exemple de scénario. On peut y voir que la mise en position marche arrière du levier de vitesse (pos) commande l'émission du signal à ultrasons sur un durée qui doit permettre d'observer un écho avec suffi une d'énergie. La période de rafraichissement de la mesure est fixée à 50 ms. Le déclenchement de l'émission du signal s'accompagne alors d'une mesure du temps de vol et d'une acquisition de l'écho éventuel.

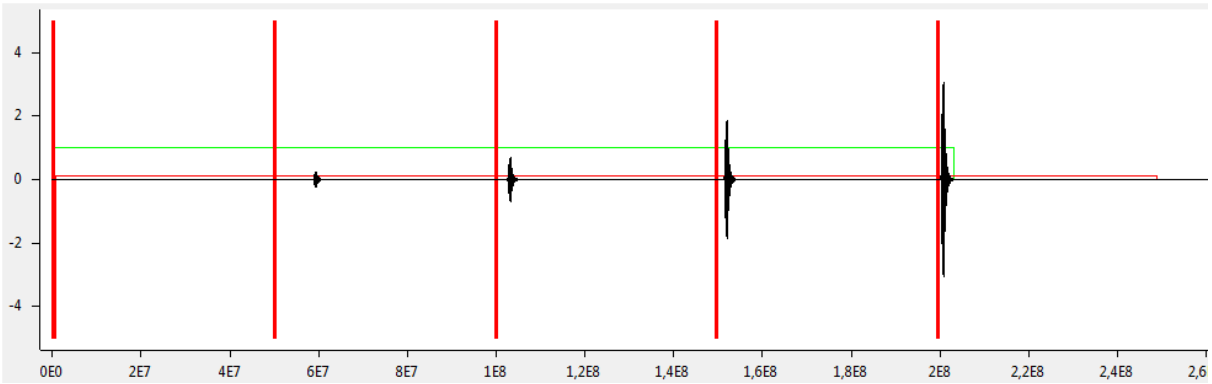


Figure 3. Séquence d'excitation et échos depuis l'émetteur/récepteur US

La figure ci-dessous montre un exemple de sorties accompagnant la réception de l'écho.

```
    Copyright (C) 2014, Intel Corporation. All rights reserved.
        SCL library release date: 2014-May-15 12:00


<INFO>:0 ns:{post-elaboration}> Monitoring manager:  parsing monitoring configuration input f
<REPORT>:11787128 ns : ctrlcapteur/traite> obstacle hors limite
<REPORT>:58821253 ns : ctrlcapteur/traite> obstacle detecté à 1.530882 metres
<REPORT>:102614503 ns : ctrlcapteur/traite> obstacle detecté à 0.511404 metres
<REPORT>:151410878 ns : ctrlcapteur/traite> obstacle detecté à 0.341044 metres
<REPORT>:200208128 ns : ctrlcapteur/traite> obstacle detecté à 0.170833 metres
[info]  Shutdown OK
```

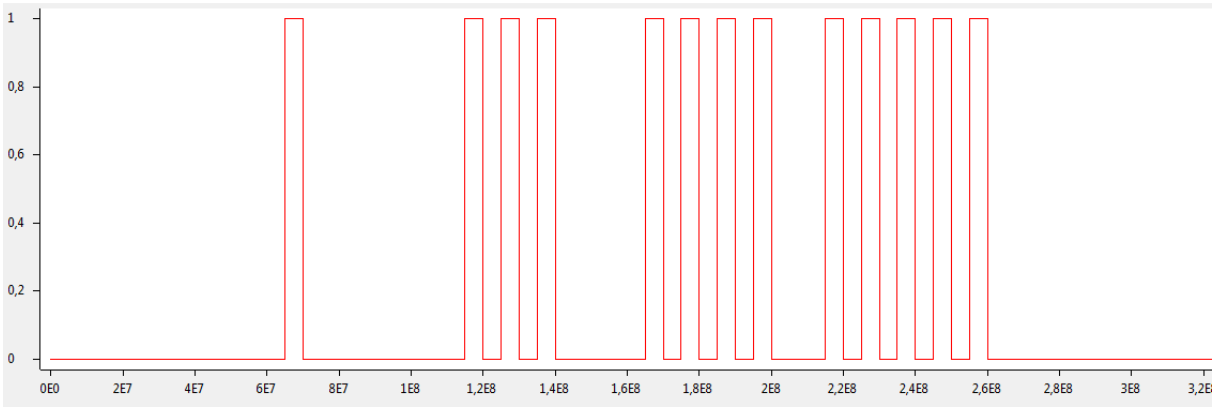


Figure 4 : Réponse sur les sorties affichage et alarme

Dans le scénario proposé, la première émission ne fournit pas d'écho mesuré avec une amplitude suffisante. On peut alors observer l'impression d'un message indiquant que la mesure est hors limite (en réalité l'affichage sera effacé). On peut ensuite s'apercevoir qu'un écho est observé, avec une amplitude croissante et avec un temps d'apparition plus faible par rapport à l'instant d'émission du *burst*. L'émission cesse lorsque le levier quitte la position de la marche arrière.

Pour la conception, les précisions suivantes sur l'application seront à prendre en compte :

1. La mesure de la distance s'effectuera sur un intervalle allant de 17 cm (zone aveugle) jusqu'à 2 mètres maximum avec une résolution du cm. L'erreur de mesure ne devra pas excéder 1 cm.

2. L'émission sonore s'effectuera sur une plage élémentaire de 1 s avec :
    - Absence de bip sonore si aucun obstacle n'est dans le champ de perception
    - 1 bip si la distance est supérieure à 1.5 m
    - 2 bips si la distance est supérieure à 1 m
    - 3 bips si la distance est supérieure à 0.5 m
    - 4 bip si la distance est supérieure à 0.25 m
    - 5 bips sinon

    On notera que l'intensité sonore sera fixe dans un premier temps mais qu'il sera possible de prévoir un ajustement de cette intensité sonore en fonction de la distance. La puissance ne devra pas dépasser les 80 db.

3. La communication entre le système de mesure de la distance et les interfaces situées au niveau de poste de conduite s'effectuera par liaison série asynchrone de type rs232C. Le protocole de communication à mettre en œuvre sera de type stop and wait avec retransmission explicite en cas d'erreur.

4. L'implémentation se fera sur une technologie de type microcontrôleur ST. On souhaite une consommation énergétique inférieure à 300 mA et une température de fonctionnement allant de -20 à 60 °C.

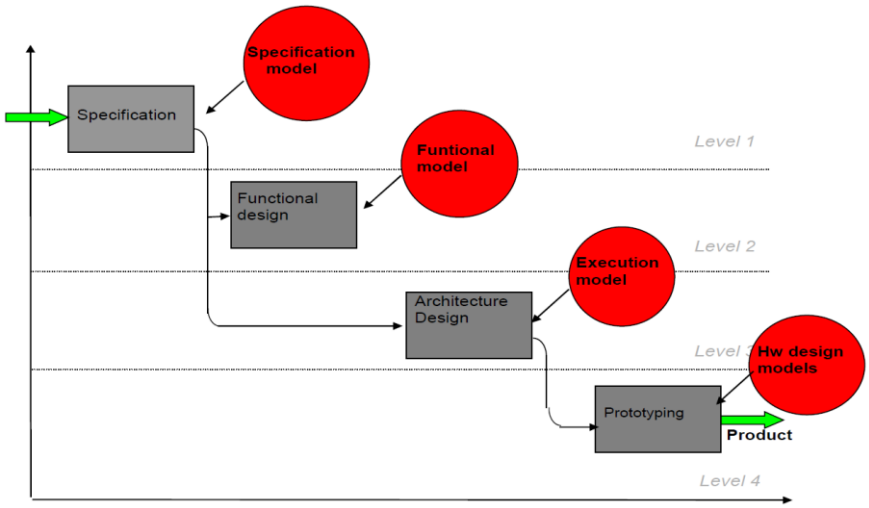


Figure 5 : Méthodologie MCSE

Le développement du projet (environ 100H) devra s'effectuer selon la méthodologie de conception des systèmes embarqués étudiée en Elec4 et rappelée sur la figure ci-dessus. Un rapport d'étude intermédiaire devra être fourni pour chaque étape de conception. Le calendrier prévisionnel sera le suivant :

- Spécifications du système :mi-septembre – fin septembre
- Etape de validation du modèle fonctionnelle sous Cofluent : Début octobre – fin octobre
- Prototypage de l'interface de communication sur ST: début octobre – fin octobre
- Etape de conception architecturale sous Cofluent : fin octobre – fin novembre
- Validation du prototype complet sur ST : fin novembre – mi-janvier

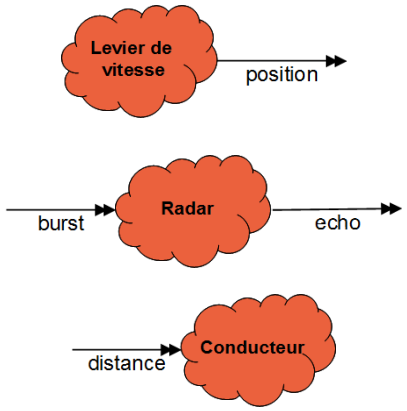
Partie 1 : Spécification du système

Objectif : L'objet de cette première partie est de préciser les spécifications du système décrit dans le cahier des charges.

A. Analyse et modélisation de l'environnement

A.1. Liste des entités

Notre environnement contient 3 entités qu'il faudra prendre en compte :

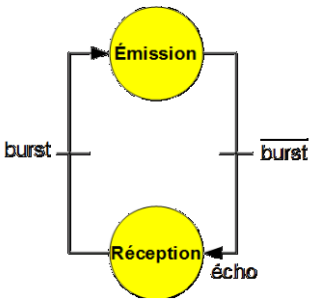


A.2. Modélisation des données

Entité	Nom du lien	Type, Sens	Définition fonctionnelle
Radar Emetteur	burst	Data, In	[boolean] Signal logique de 0 à 5V, Fréquence 40kHz.
Radar Récepteur	echo	Data, Out	Signal analogique de -2.5V à 2.5V.
Levier de vitesse	position	Data, Out	[boolean] Signal logique de 0 à 5V.
Conducteur	distance	Data, In	[17cm...200cm,float] Distance mesurée de 17cm à 2m.

A.3. Modélisation du comportement

A.3.1 Radar



Nous modéliserons l'entité sous Matlab.

a. Onde burst

On suppose 2 types d'ondes $burst$ possibles : sinusoïdale ou carrée, comprises entre 0 et 5V. La fréquence du signal est de 40kHz.

b. Onde Vsconv émise

Le comportement de l'émetteur ultrason peut être modélisé par une fonction de transfert du second ordre du type :

$$H(s) = \frac{X(s)}{Y(s)} = \frac{K}{s^2 + 2\omega_0\zeta s + {\omega_0}^2}$$

On applique une transformation en z par la méthode de Tustin, où $s = \frac{2}{T}\frac{1-z^{-1}}{1+z^{-1}}$.

Ainsi, $H(z) = \frac{K \quad + \quad 2K.z^{-1} \quad + K.z^{-2}}{\left(\frac{4}{T^2}+\frac{4\omega_0\zeta}{z}+\omega_0^2\right)+z^{-1}.\left(2\omega_0^2-\frac{8}{T^2}\right)+z^{-2}.\left(\frac{4}{T^2}+\omega_0-\frac{4\omega_0\zeta}{T}\right)}$ il est alors possible d'utiliser la fonction *filter* et de l'appliquer au signal d'entrée $burst$.

On considère pour la simulation une fréquence d'échantillonnage de 800kHz afin d'étudier les ondes émises et reçues avec plus de précision, mais il conviendra de revoir cette fréquence à la baisse pour le système réel.

c. Onde reçue par le récepteur

Nous avons 2 méthodes de calcul de l'atténuation :

Métode 1 : Formule de Friis

Nous utilisons la formule de Friis pour les télécommunications afin d'évaluer l'atténuation du signal reçu par le récepteur :

$$\frac{P_r}{P_t} = G_t G_r \frac{\lambda^2}{(4\pi R)^2} = \frac{\left(\frac{c}{f}\right)^2}{(4\pi R)^2} = G_t G_r \frac{\left(\frac{340}{40.10^3}\right)^2}{(4\pi.2d)^2} = G_t G_r \frac{114.10^{-9}}{d^2}$$

Avec *d*, la distance de l'obstacle par rapport à l'émetteur/récepteur, G_t le gain de l'antenne d'émission, et G_r le gain de l'antenne de réception. Nous devrons préciser les gains G_t et G_r avant modélisation précise de l'atténuation du signal.

Métode 2 : Courbe d'atténuation dans l'air en fonction de la fréquence et de l'humidité ambiante

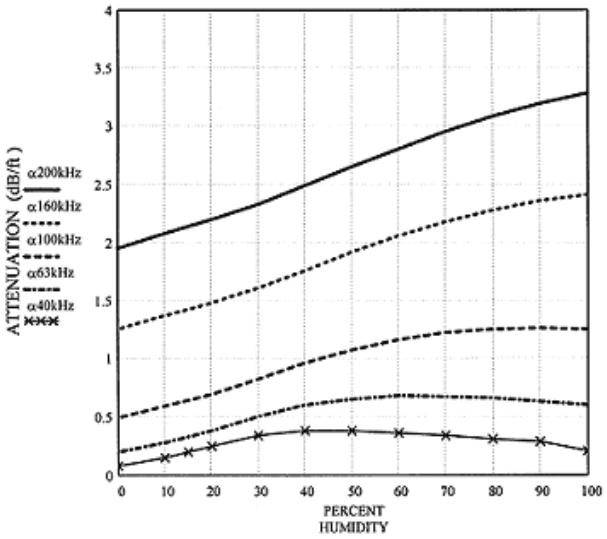


Figure 1 - Cette série de courbes montre les variations de l'atténuation du son dans l'air en fonction de l'humidité, pour des fréquences comprises entre 40kHz et 200 kHz

Sur cette courbe, on remarque qu'à une fréquence de 40kHz, l'atténuation est inférieure à 0.5dB/ft, soit 1.65dB/m. Nous pourrons arrondir à une valeur de 1.5dB/m dans les conditions normales d'utilisation. Ainsi, si l'obstacle est placé à 2m du radar (valeur maximum), l'atténuation répartie sur 4m atteint 6dB : $P_{dB\ reçue} = P_{dB\ émise} - 6dB \quad \Rightarrow \quad V_r = V_e . 10^{\frac{-6}{20}} = \frac{5}{2} = 2{,}5V$

d. Modélisation Matlab

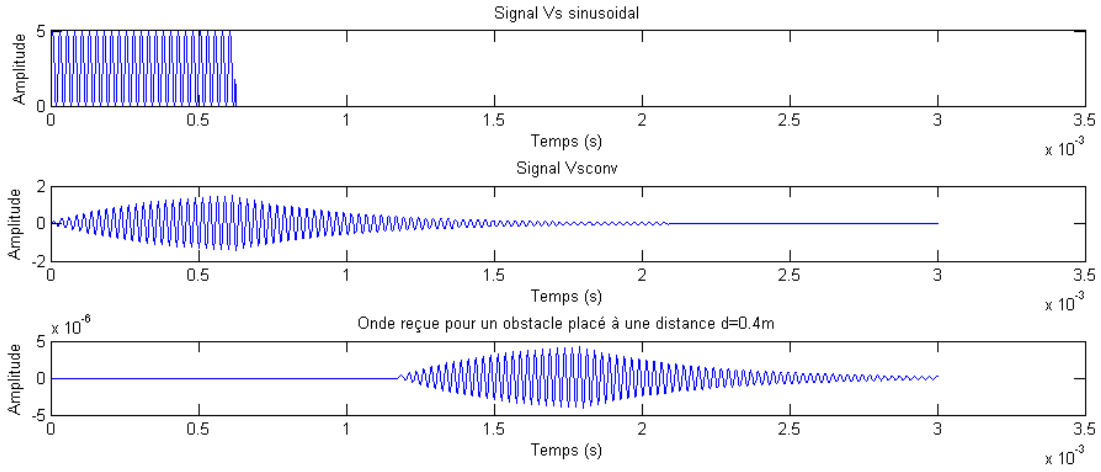


Figure A.2 - Signaux en réception et en émission pour un signal burst *sinusoïdal*

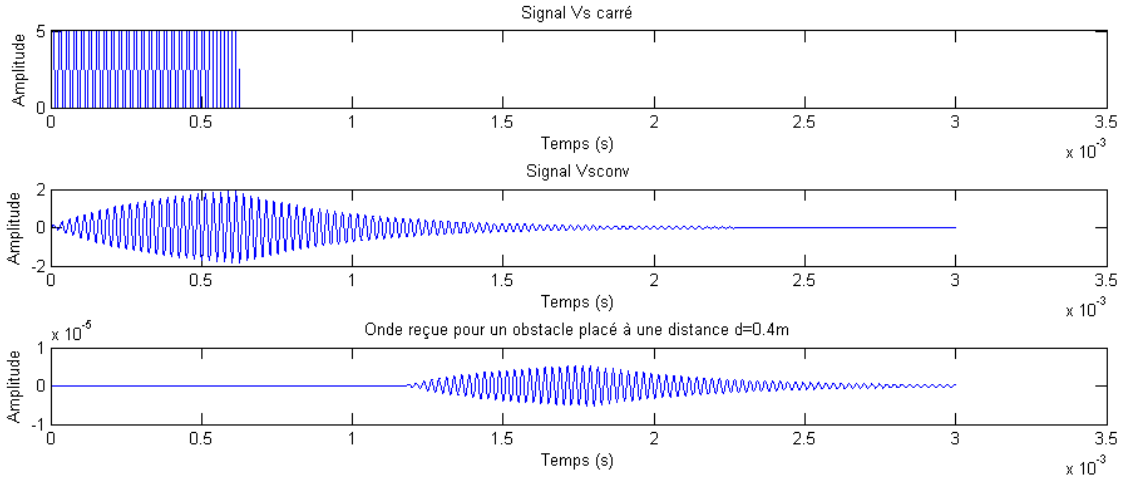
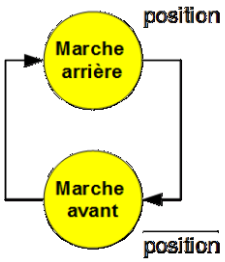


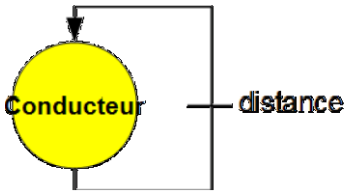
Figure A.3- Signaux en réception et en émission pour un signal burst *sinusoïdal*

La simulation de l'entité sous Matlab donne une onde émise et réfléchie globalement similaire pour un signal *burst* sinusoïdal ou carré. **Nous pourrons donc opter pour un signal *burst* carré évitant l'utilisation du convertisseur numérique-analogique du contrôleur.**

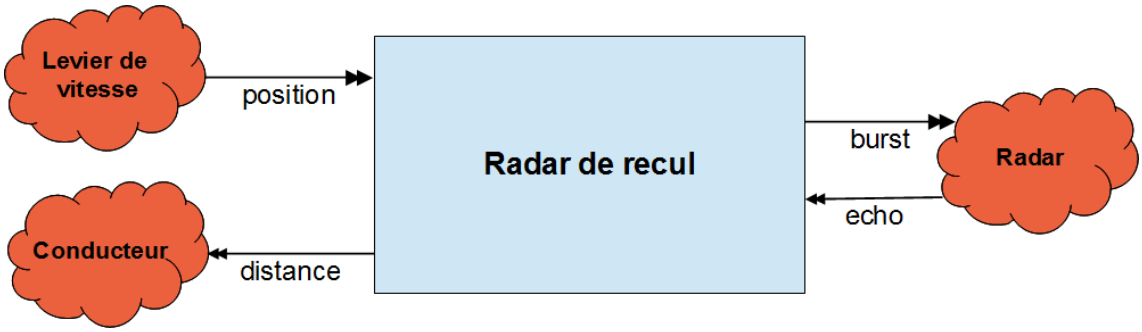
A.3.2 Levier



A.3.3 Conducteur



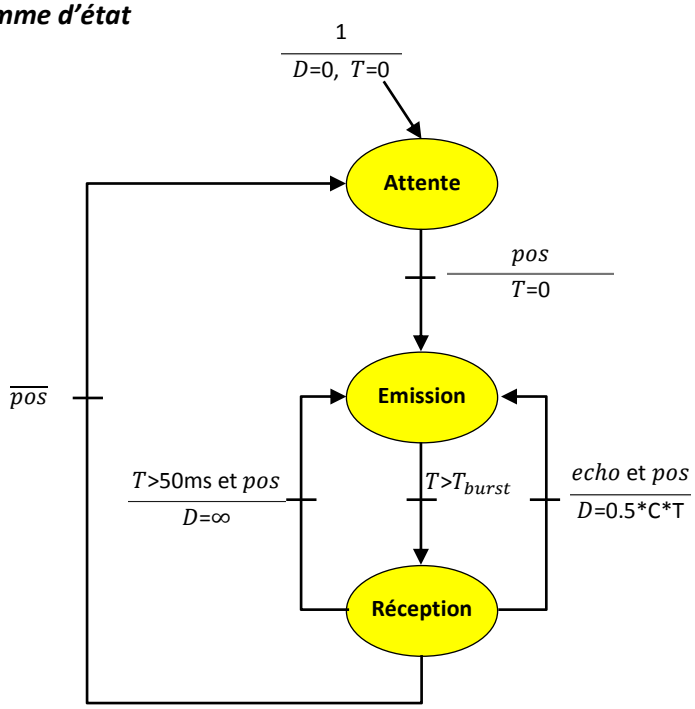
A.4. Diagramme de contexte



B. Spécifications

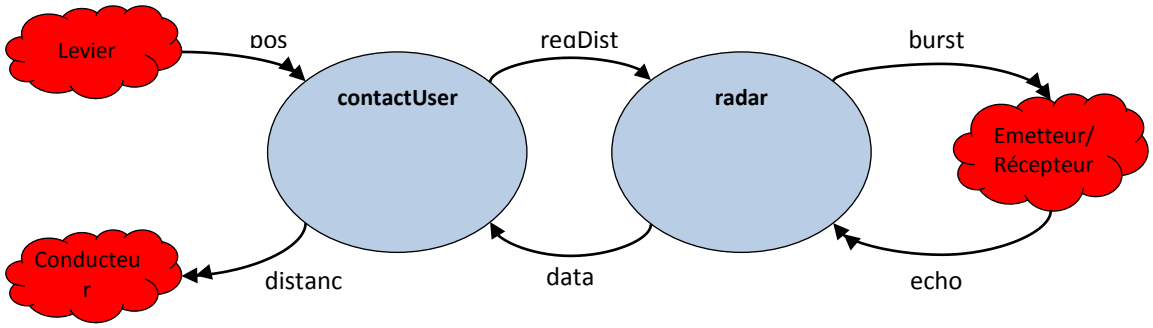
B.1. Spécifications fonctionnelles

B.1.1 Diagramme d'état



Nous utilisons une approche par les entrées : nous déterminons, à partir d'un état initial, les valeurs des sorties $burst$ et $distance$, suite au changement d'état des entrées pos et $echo$.

B.1.2 Diagramme des activités



B.1.3 Modélisation des actions internes

Nom du lien	Type	Définition fonctionnelle
reqDist	Event	Demande de distance de la part du superviseur vers le contrôle.
data	Information	Envoi d'un événement signifiant qu'une distance est disponible.

B.2. Spécifications opératoires

Signal $burst$: Signal carré de fréquence $f = 40kHz$

T_{burst} : 25 périodes du signal $burst$, soit $T_{burst} = \frac{25}{40.10^3} = 625\mu s$

D : Distance calculée par le radar : $D = \frac{C.T}{2}$ avec C=344 m/s

$seuil$: Seuil de détection de l'onde reçue d'après la modélisation de l'atténuation. Le convertisseur Analogique Numérique dispose d'une résolution minimale de 10 bits, soit une sensibilité à

B.3. Spécifications technologiques et économiques

- Buzzer pouvant émettre jusqu'à 80dB. Durée du $burst$ à définir suivant l'énergie nécessaire et de la zone aveugle. Intervalle entre les $burst$ de 50ms.

- Afficheur LCD

- Micro-contrôleur ST

- Liaison RS232 entre le radar et les interfaces

- Emetteur / Récepteur ultrason, avec une erreur maximum de 1cm.

Conclusion de la partie 1

L'environnement et les spécifications ainsi précisées, nous devons maintenant trouver une solution d'implémentation du modèle en le modélisant et le simulant progressivement sous *Cofluent* afin de le faire valider.

Partie 2 : Validation du modèle fonctionnel

Objectif : L'objet de cette seconde partie est de donner une solution d'implémentation du modèle décrit lors de la première étape (spécification du système) en terme de fonctions et de relations, et simuler son fonctionnement sous Cofluent.

A. Modélisation de l'émetteur/récepteur

L'émetteur/Récepteur peut être modélisé de manière indépendante en supposant un message d'entrée de type $burst$: signal carré sur 5 périodes puis reste à 0.

Le modèle de l'ER utilise la fonction de transfert utilisée en partie 1, et introduit un retard proportionnel à la distance, ainsi qu'une atténuation.

B. Données de modélisation

Fonction de transfert

$$H(z) = \frac{K \quad + \quad 2K.z^{-1} \quad + \quad K.z^{-2}}{\left(\frac{4}{T^2} + \frac{4\omega_0\zeta}{z} + \omega_0^2\right) + z^{-1}.\left(2\omega_0^2 - \frac{8}{T^2}\right) + z^{-2}.\left(\frac{4}{T^2} + \omega_0 - \frac{4\omega_0\zeta}{T}\right)}$$

On considère pour la simulation une fréquence d'échantillonnage de 800kHz afin d'étudier les ondes reçues avec plus de précision, mais il conviendra de revoir cette fréquence à la baisse pour le système réel.

Equation de récurrence

$$s(n) = \frac{1}{a}[K.e(n) + 2K.e(n-1) + K.e(n-2) - b.s(n-1) - c.s(n-2)]$$

Avec :

$$\begin{cases} a = \frac{4}{\mathrm{Te}^2} + 4.\omega * \frac{\zeta}{\mathrm{Te}} + \omega^2 = 2.6312.10^{12} \\ b = 2.\omega^2 - \frac{8}{\mathrm{Te}^2} = -4.9937.10^{12} \\ c = \frac{4}{\mathrm{Te}^2} + \omega - 4.\omega * \frac{\zeta}{\mathrm{Te}} = 2.6151.10^{12} \\ K = 5.10^6 \\ \zeta = 0.1 \end{cases}$$

Retard

$$t_{retard} = \frac{2 * dist}{v_{son}} = \frac{2 * dist\ (cm)}{34\ 000\ (cm/s)}$$

Le retard est modélisé par une FIFO initialisée à 0 dont la taille est proportionnelle à la distance. Ainsi, le signal echo en sortie sera à 0 le temps que la fifo se « remplisse » des valeurs obtenues par la fonction de transfert, et cela pour une durée équivalente à t_{retard}, soit en terme d'échantillons : $t_{retard} * F_e$.

On a donc :

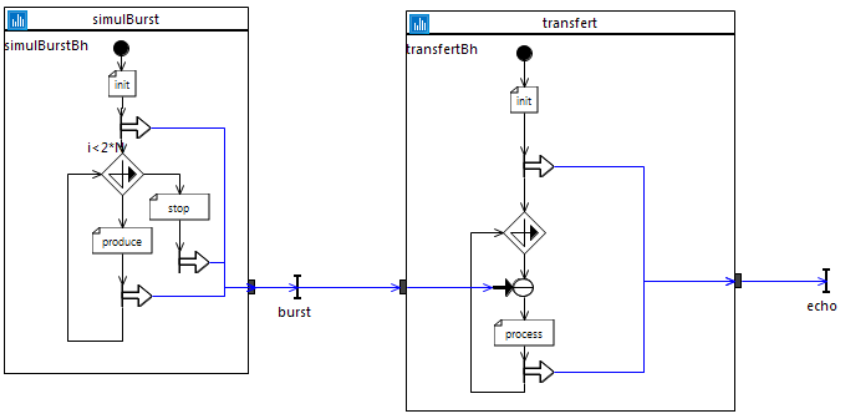
$$tailleFIFO = \frac{2 * dist}{34000} * F_e$$

Atténuation

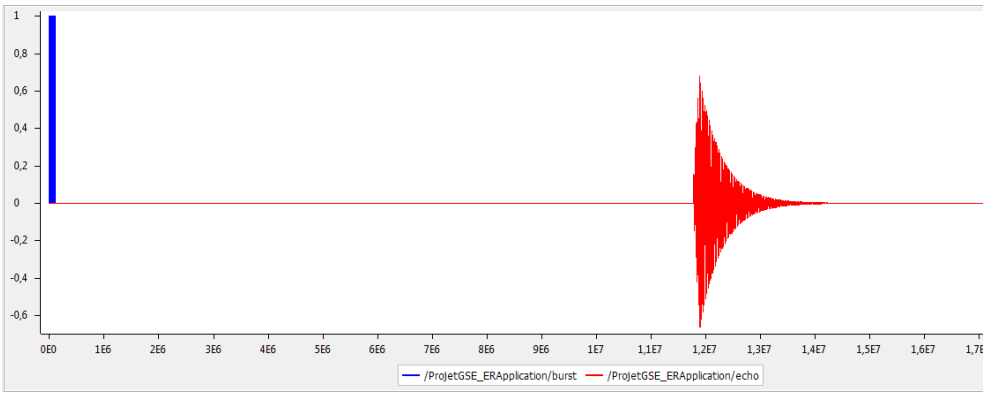
Les différentes formules donnant l'atténuation d'un signal ultrason dans l'air à courte distance n'étant pas suffisamment réalistes dans notre cas, nous avons choisi de considérer une atténuation de 50% à 1m, puis 75% à 2m, etc. La formule utilisée est la suivante :

$$att = 0.5^{(\frac{dist}{100})}$$

B.1. Modélisation Cofluent



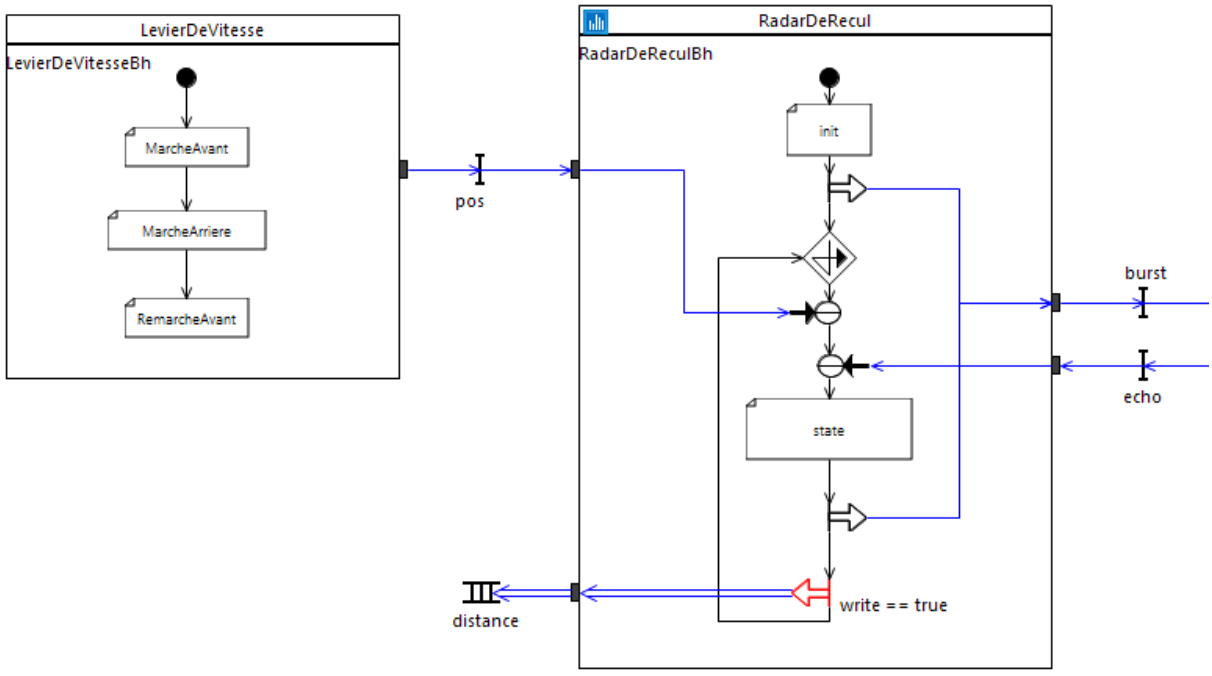
B.2. Simulation



La simulation affiche en bleu le signal $burst$ généré en entrée, puis l'on observe un silence en sortie pour une durée d'environ 11.8 ms, correspondant à une distance de $\frac{t_{retard} * v_{son}}{2} = 2.006$ mètres, avant d'afficher l'echo du signal $burst$.

C. Modélisation de la machine à états

Le système est modélisé par une machine à état comme cela a été défini dans l'étude préliminaire.

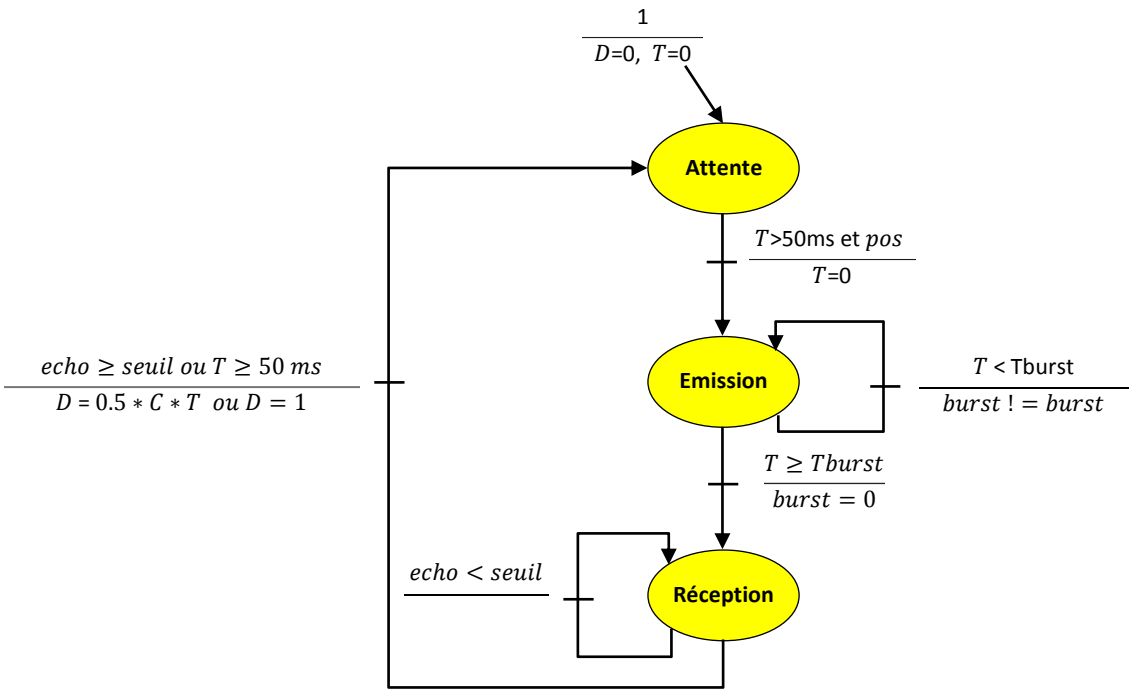


La machine à état est réalisée dans un bloc fonctionnel « Sampled » avec une période de T_{pas} soit 12.5 µs. Cette période correspond au plus petit temps utilisé dans le système.

De plus, le bloc comprend une étape d'initialisation. Les variables $echo$, pos et $burst$ seront actualisées à chaque passage dans la machine à état. Seule la variable $distance$ sera écrite uniquement en fin de réception.

```
1 void cfm_op_init()
2 {
3     etat = idle;
4     t = 0.0;
5     //distance = 0;
6     burst = 0;
7     write = false;
8     cf_trace_report("Init_RadarDeRecul");
9 }
```

La machine à état est composée de 3 états : un état d'attente, un d'émission et un de réception.



- **Attente** : Le système reste dans cet état tant que le véhicule est en marche avant, ou en attendant la prochaine occurrence d'envoi du signal $burst$ (50ms).

- **Emission** : Cet état génère le signal $burst$ durant T_{burst} (125 µs) . Ce signal est un signal carré de période 2*T_{pas} (2*12.5 µs = 25 µs).

- **Réception** : L'état de réception attend que le signal $echo$ soit supérieur au seuil pour calculer la distance en fonction de temps écoulé depuis le début d'émission. Dans le cas où ce temps dépasse 50 ms, nous passons à l'état suivant en indiquant une distance de 1 pour indiquer que la distance est hors limite.

```
void cfm_op_state()
{

    switch(etat){
        case idle :
            if(pos == true and t>=Tglobal){
                cf_trace_report("Idle -> emission");
                t = 0;
                write = false;
                distance = 0.0;
                etat = emission;
            }break;

        case emission :
            if(t<=Tburst){
                burst = (burst==0)?1:0;
            }
            else{
                cf_trace_report("Emission -> Reception");
                burst = 0;
                etat = reception;
            }break;

        case reception :
            if(t>=Tglobal){
                distance = 1.0;
                write = true;
                etat = idle;
                cf_trace_report("Reception -> Idle OOR %f",t);
            }
            else if(echo>=EchoSeuil){
                distance = 0.5*C*t;
                write = true;
                cf_trace_report("Reception -> Idle avec distance = %f at %f", distance,t);

                etat = idle;

            }break;
        default :
            cf_trace_report("Etat error");
    }
    t = t+Tpas;
}
```

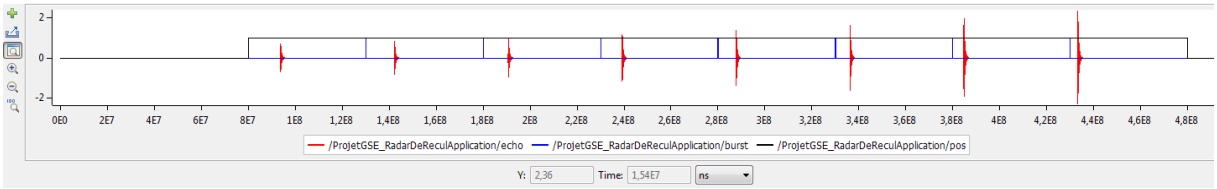
D. Simulation Machine à état + Emetteur/Récepteur

Nous regroupons maintenant notre machine à état avec l'émetteur/récepteur afin de simuler le fonctionnement de la machine à état.

Pour cela, nous ajoutons le deux producteurs :

- Le levier de vitesse qui va simuler une mise en marche arrière puis un retour en marche avant.
- L'environnement qui va fournir une distance variable à l'émetteur/récepteur.

Voici une simulation pour une distance variant de 230 cm à 30 cm par pas de 25 cm.



Nous pouvons valider un grand nombre de points avec cette simulation :

- **Levier de vitesse** : Nous constatons, que le système réalise des mesures uniquement lors d'une marche arrière.
- **Burst** : Les $burst$ sont bien envoyés à intervalle régulier de 50 ms. De plus, la séquence de $burst$ envoie comme souhaité 5 $burst$ de 12,5 µs.
- **Echo** : Le signal $echo$ correspond aux simulations de l'émetteur/récepteur seul.
- **Distance** : Notre machine à état calcule bien pour chaque mesure la distance envoyée par le producteur avec une erreur inférieure à 1 cm. L'erreur est due à la fréquence d'échantillonnage. Soit une erreur de distance maximum de C*Tpas soit 0.425 cm.

Après avoir validé le fonctionnement de notre machine à état, nous allons ajouter la communication stop and wait vue l'année précédente.

E. Communication Stop and Wait

Dans cette partie, nous allons inclure dans notre projet la communication orientée caractère en mode stop and wait vue l'année précédent. Ce système de communication se place en sortie du radar de recul et capte la valeur stockée dans le Message Queue *distance*, afin de la transmettre à un consommateur. Le protocole utilisé se charge de la vérification du bit de parité lors de l'émission de chaque caractère, de la vérification du Checksum en fin de transmission, et de l'acquittement ou non de la séquence. Si un problème survient, la séquence est retransmise jusqu'à un total de 3 essais au bout desquels un signal d'échec sera envoyé au système.

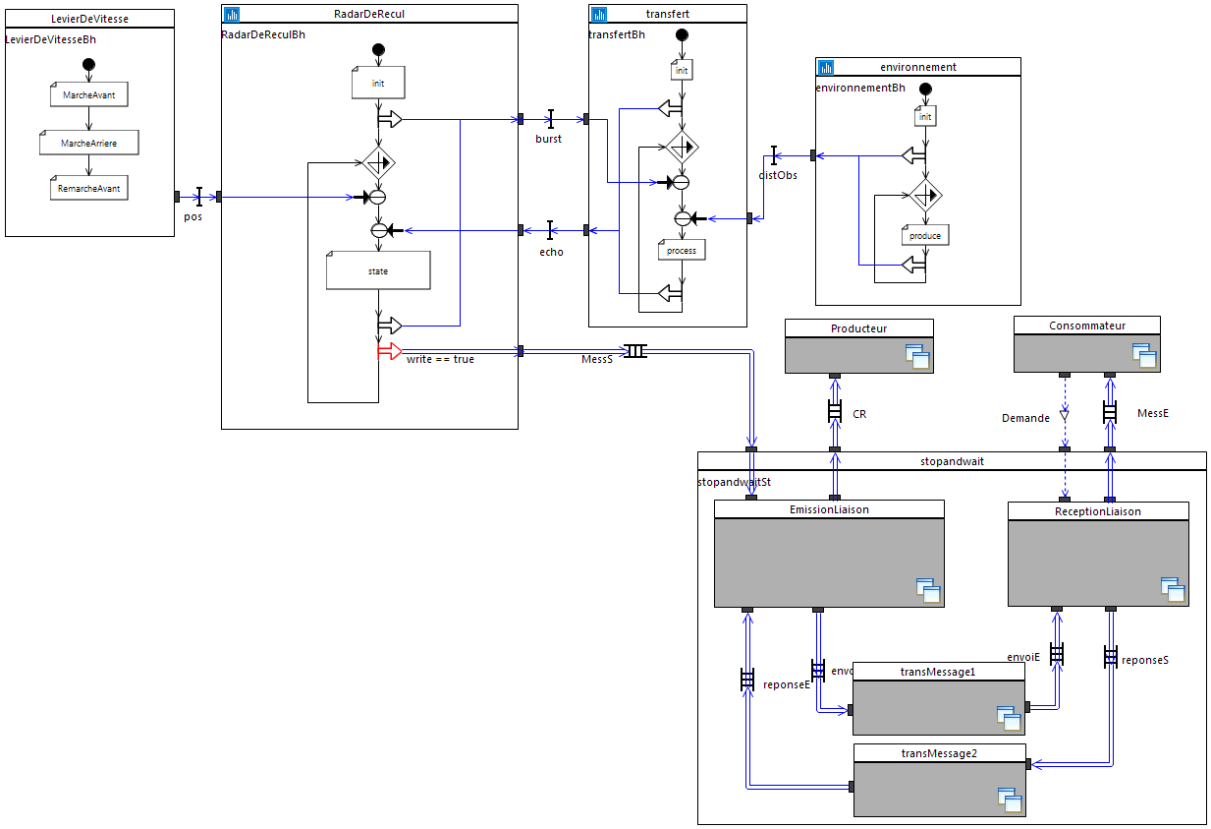
E.1. Première solution d'implémentation dans le modèle

Ici, le producteur du message est compris dans le radar de recul, le consommateur est un bloc à part, et la fonction de transmission stop&wait constitue un bloc bien défini.

Nous avons donc dû adapter le radar de recul pour fonctionner en mode *producteur* : il émet la distance non-plus sous forme d'un *float* ou d'un entier, mais sous forme de caractères formant une chaine : MessS. Il s'occupe également de recevoir la confirmation de réception (signal CR) pour laquelle nous devons encore préciser notre stratégie d'action en cas d'erreur : soit nous tenons compte des 3 échecs successifs et nous tentons de relancer l'envoi, soit nous ignorons l'erreur et avançons dans la machine à état.

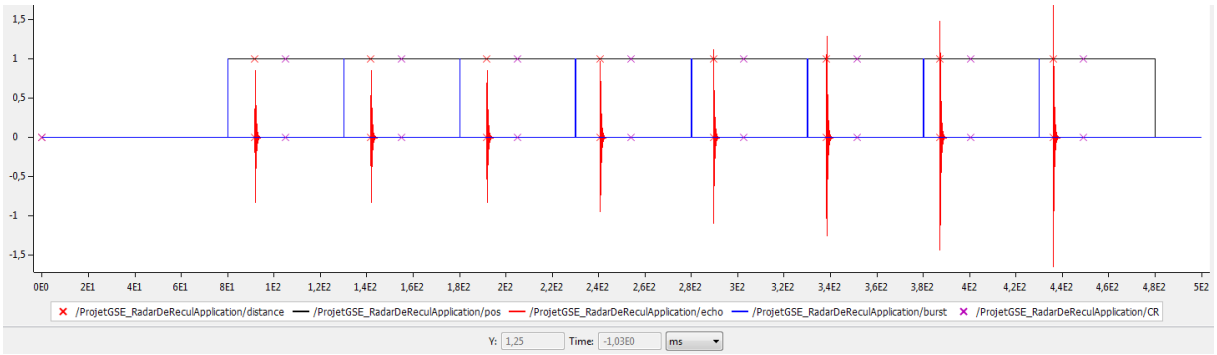
E.2. Modélisation du système

Sur cet aperçu, on retrouve à gauche le levier de vitesse et le radar de recul contenant la machine à états, puis à sa droite le bloc modélisant l'émetteur/récepteur ainsi que le bloc modélisant l'environnement (distance de l'obstacle), et en dessous le bloc de transmission stop&wait.



E.3. Simulation

La simulation de notre système nous permet de visualiser la position du levier de vitesse, les variations des signaux $burst$ et $echo$, l'état de la FIFO $distance$, et l'état de la confirmation de réception CR, lorsque la voiture s'approche d'un obstacle.



Analyse du graphe :

- La première observation évidente est l'augmentation de la puissance de l'écho reçu (signal rouge continu): le signal émis étant supposé de 3,3V nous obtenons bien un écho atténué de 50% lorsque l'obstacle est proche de 1m (dernier écho reçu sur la simulation).
- Lorsqu'un écho est reçu, la FIFO *distance* (croix rouges) reçoit la distance mesurée et se place donc dans un état « plein » à 1.
- En fin de transmission, on observe la présence d'une confirmation de réception CR (croix violettes).
- Le signal $burst$ est bien envoyé toutes les 50ms
- Aucun signal n'est manipulé en dehors de la position « marche arrière ».

Nous vérifions le protocole de transmission grâce à la fonction cf_trace_report placé à chaque étape, ici en envoyant la valeur ''120'' ==> [STX][49][50][48][CS][ETX] :

```
<REPORT>:387087522 ns : RadarDeRecul/state> Reception -> Idle avec distance = 120.487500
<REPORT>:387087547 ns : CtrlEmission/envoi> Nouveau message emis : 120
<REPORT>:387087571 ns : Transmetteur/test1> Emetteur --> STX
<REPORT>:388275013 ns : recepteur/test3> Recepteur :  STX
<REPORT>:388425025 ns : Transmetteur/test1> Emetteur --> caractère : 49
<REPORT>:389625013 ns : recepteur/test3> Recepteur : Caractère reçu : 49
<REPORT>:389775025 ns : Transmetteur/test1> Emetteur --> caractère : 50
<REPORT>:390975013 ns : recepteur/test3> Recepteur : Caractère reçu : 50
<REPORT>:391125025 ns : Transmetteur/test1> Emetteur --> caractère : 48
<REPORT>:392325013 ns : recepteur/test3> Recepteur : Caractère reçu : 48
<REPORT>:392475025 ns : Transmetteur/test1> Emetteur --> caractère : 103
<REPORT>:393675013 ns : recepteur/test3> Recepteur : Caractère reçu : 103
<REPORT>:393825025 ns : Transmetteur/test1> Emetteur --> ETX
<REPORT>:395025013 ns : recepteur/test3> Recepteur : ETX
<REPORT>:395025050 ns : Consommateur/op> Consommateur : Message recu : 120
<REPORT>:395025061 ns : Transmetteur/test1> Emetteur --> STX
<REPORT>:396212513 ns : recepteur/test3> Recepteur :  STX
<REPORT>:396362525 ns : Transmetteur/test1> Emetteur --> caractère : 6
<REPORT>:397562513 ns : recepteur/test3> Recepteur : Caractère reçu : 6
<REPORT>:397712525 ns : Transmetteur/test1> Emetteur --> caractère : 244
<REPORT>:398912513 ns : recepteur/test3> Recepteur : Caractère reçu : 244
<REPORT>:399062525 ns : Transmetteur/test1> Emetteur --> ETX
<REPORT>:400262513 ns : recepteur/test3> Recepteur : ETX
<REPORT>:400262549 ns : Producteur/Operation> CR = Succès
```

Un signal CR=success est bien retourné au producteur en fin de transmission.

De même, nous obtenons un rapport d'envoi successif des distances mesurées (de 160cm à 100cm) grâce à la fonction cf_trace_report placée à plusieurs positions stratégiques :

```
<REPORT>:280000022 ns : RadarDeRecul/state> Burst
<REPORT>:289437522 ns : RadarDeRecul/state> Reception -> Idle avec distance = 160.437500
<REPORT>:289437547 ns : CtrlEmission/envoi> Nouveau message emis : 160
<REPORT>:297375050 ns : Consommateur/op> Consommateur : Message recu : 160
<REPORT>:302612549 ns : Producteur/Operation> CR = Succès
<REPORT>:330000022 ns : RadarDeRecul/state> Burst
<REPORT>:338262522 ns : RadarDeRecul/state> Reception -> Idle avec distance = 140.462500
<REPORT>:338262547 ns : CtrlEmission/envoi> Nouveau message emis : 140
<REPORT>:346200050 ns : Consommateur/op> Consommateur : Message recu : 140
<REPORT>:351437549 ns : Producteur/Operation> CR = Succès
<REPORT>:380000022 ns : RadarDeRecul/state> Burst
<REPORT>:387087522 ns : RadarDeRecul/state> Reception -> Idle avec distance = 120.487500
<REPORT>:387087547 ns : CtrlEmission/envoi> Nouveau message emis : 120
<REPORT>:395025050 ns : Consommateur/op> Consommateur : Message recu : 120
<REPORT>:400262549 ns : Producteur/Operation> CR = Succès
<REPORT>:430000022 ns : RadarDeRecul/state> Burst
<REPORT>:435912522 ns : RadarDeRecul/state> Reception -> Idle avec distance = 100.512500
<REPORT>:435912547 ns : CtrlEmission/envoi> Nouveau message emis : 100
<REPORT>:443850050 ns : Consommateur/op> Consommateur : Message recu : 100
<REPORT>:449087549 ns : Producteur/Operation> CR = Succès
```

De même, un signal CR=success est bien retourné au producteur à la fin de chaque transmission, et le consommateur peut effectivement disposer des valeurs de distance toutes les 50ms.

Conclusion de la partie 2

Cette étape de modélisation et de simulation permet de raffiner notre modèle et d'y apporter les précisions nécessaires à son bon fonctionnement. Nous obtenons des résultats cohérents lors des simulations et bien que tous les cas de figure ne soient pas présentés dans ce rapport, nous nous attachons à tester un maximum de cas limites et à vérifier que le système répond de façon non-bloquante.

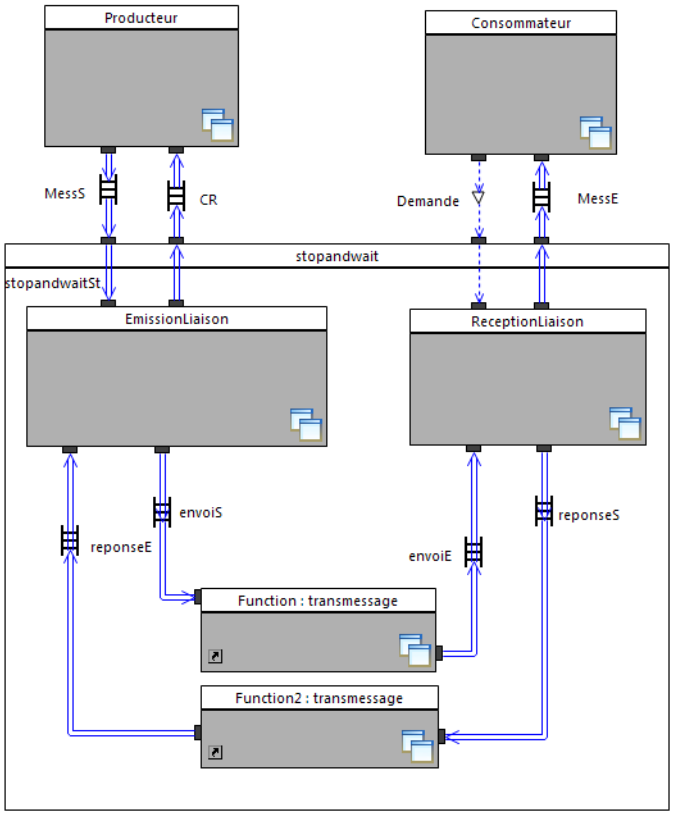
Partie 3 : Communication, et interfaces : modélisation

Objectif : L'objet de cette troisième partie du projet est notamment de préciser la méthode de communication interne au système et la gestion des interfaces. Nous allons donc étudier le protocole de communication orienté caractère, et prendre en compte les interfaces en modélisant notamment leur fonctionnement et leur gestion.

A. Solution fonctionnelle avec protocole de communication

A.1. Modèle de la transmission

Nous nous appuyons sur l'interface de communication *Stop&Wait* modélisée l'année dernière sur *Cofluent*, que nous allons corriger, sécuriser, et adapter à notre système.



Dans cette communication, la couche supérieure appelée « Liaison » se charge de transmettre le message côté producteur et de le réceptionner et l'acquitter côté consommateur. Si un NACK est renvoyé ou si le délai de réponse dépasse le Timeout, la fonction « EmissionLiaison » se charge de le renvoyer jusqu'à 3 fois.

La couche située en dessous appelée « Framing/Deframing » se charge de l'encapsulation du message avec un caractère de début de trame, de fin de trame, et en plaçant un caractère de Checksum en avant-dernière position. 3 caractères sont donc ajoutés et permettent lors du *Deframing*, de déterminer parfaitement les bornes du messages et de vérifier la bonne transmission en recalculant le *Checksum*. Si une erreur intervient, un NACK sera retourné.

Enfin, la couche du niveau inférieure, appelée « Transmission Caractère » se charge de décomposer et reconstruire chaque caractères en suite binaire, en incorporant un bit de signe pour vérifier la validité du caractère transmis.

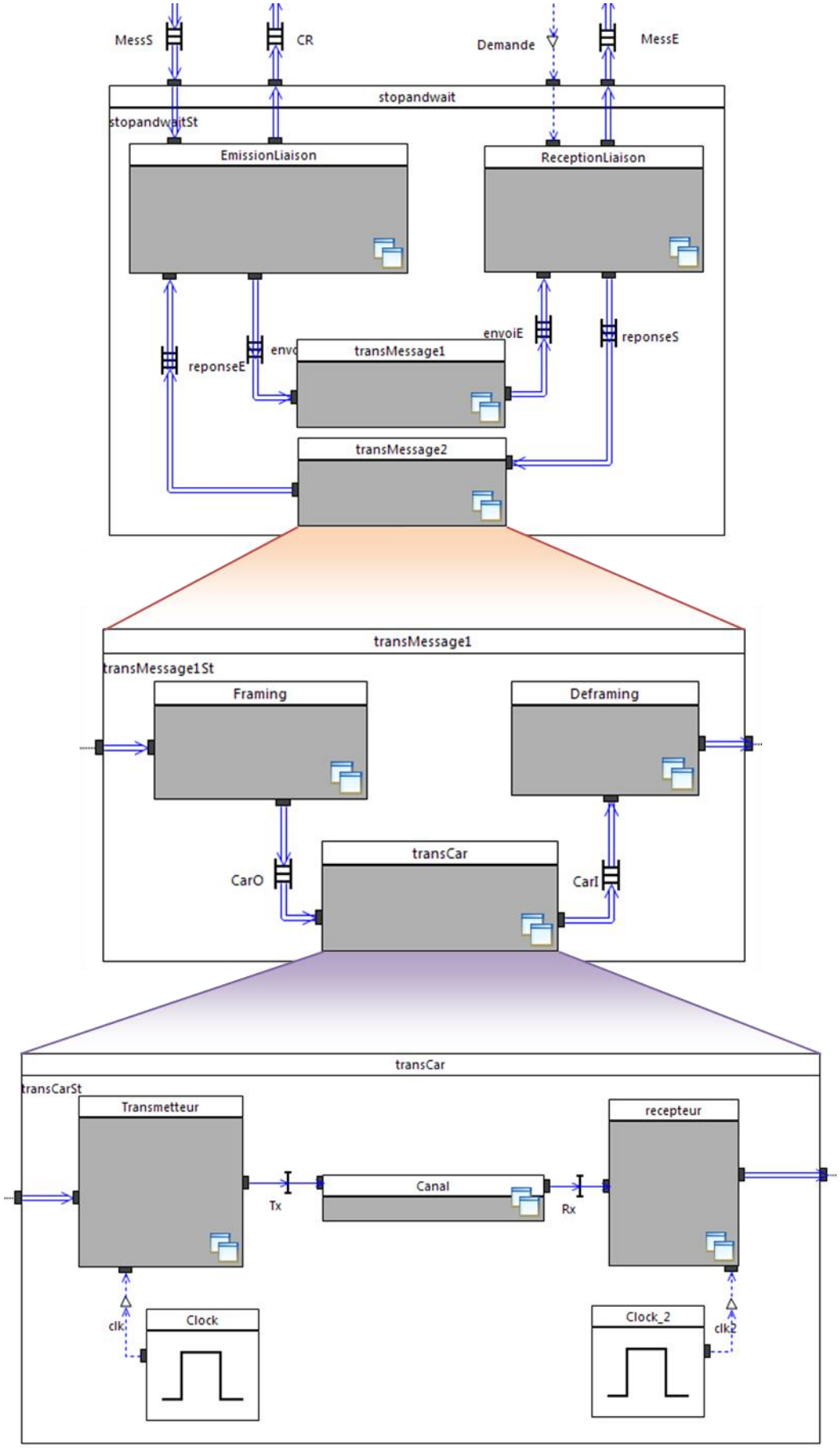
Au niveau encore inférieur nous trouvons le canal, dans lequel du bruit peut survenir.

(Illustration de la structure page suivante)

Dans notre système, le message provient de notre fonction « RadarDeRecul » qui devient alors le producteur. L'information est transmise sous forme de chaîne de caractères au gestionnaire d'interfaces qui se charge de la gestion du Buzzer et de l'écran.

Cependant, nous pouvons noter qu'une difficulté peut se présenter sur ce modèle de communication : elle est unilatérale et ne permet pas au gestionnaire d'interface de communiquer des informations au radar de recul. Les messages d'acquittement et de non-acquittement sont générés automatiquement par la couche *Liaison* de la structure *Stop&Wait* et cela ne permet pas au consommateur de prendre la main. Nous choisissons donc de conserver la connexion du signal pos du levier de vitesse à la fonction RadarDeRecul afin de ne pas nécessiter de modification du protocole.

Une seconde observation que nous pouvons faire est que le cahier des charges ne précise pas la réponse du système en cas d'échecs successifs de transmission des données, c'est pourquoi nous avons choisi d'intercepter cette information d'échec (CR=echec) comme condition de sortie de la boucle principale de la fonction RadarDeRecul, entrainant l'arrêt des séquences de mesure sur le capteur de distance. Cela permet d'éviter une consommation de la plateforme dans une situation de disfonctionnement, mais impose un redémarrage pour reprendre un fonctionnement normal. Nous faisons donc le choix de la maintenance manuelle afin de favoriser l'économie d'énergie.


MessS
CR
Demande
MessE
stopandwait
stopandwaitSt
EmissionLiaison
ReceptionLiaison
reponseE
envoiE
reponseS
transMessage1
transMessage2
transMessage1
transMessage1St
Framing
Deframing
transCar
CarO
CarI
transCar
transCarSt
Transmetteur
recepteur
Canal
Tx
Rx
clk
Clock
Clock_2
clk2

A.2. Couche supérieure « Liaison » : EmissionLiaison / RéceptionLiaison

A.2.1 EmissionLiaison

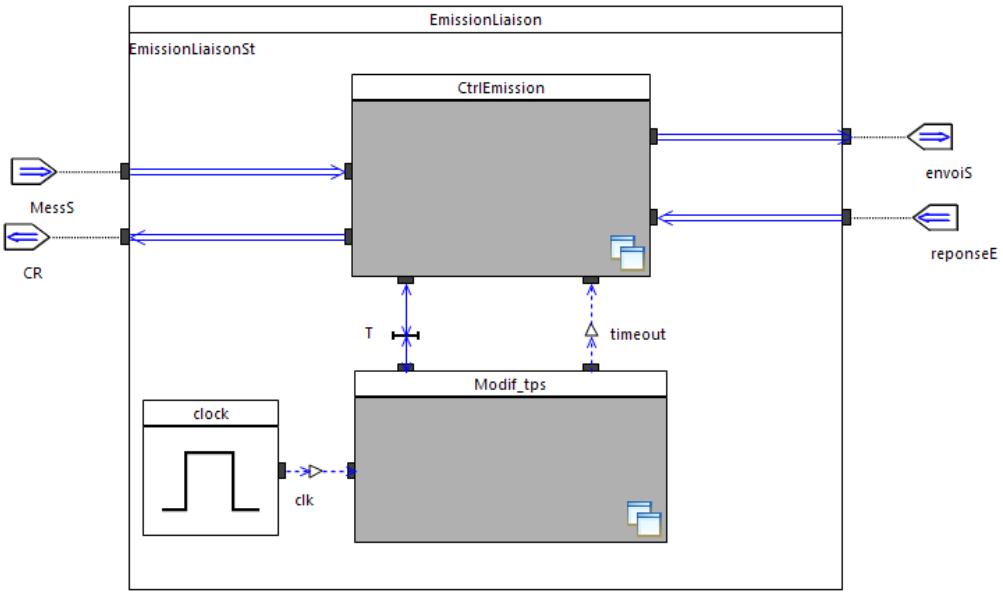


Figure 4 - Schéma du bloc EmissionLiaison

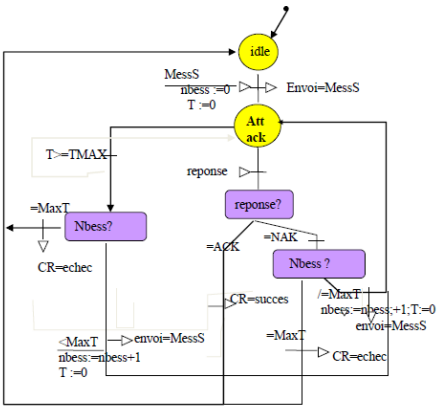


Figure 5 - Fonctionnement attendu du contrôleur d'émission

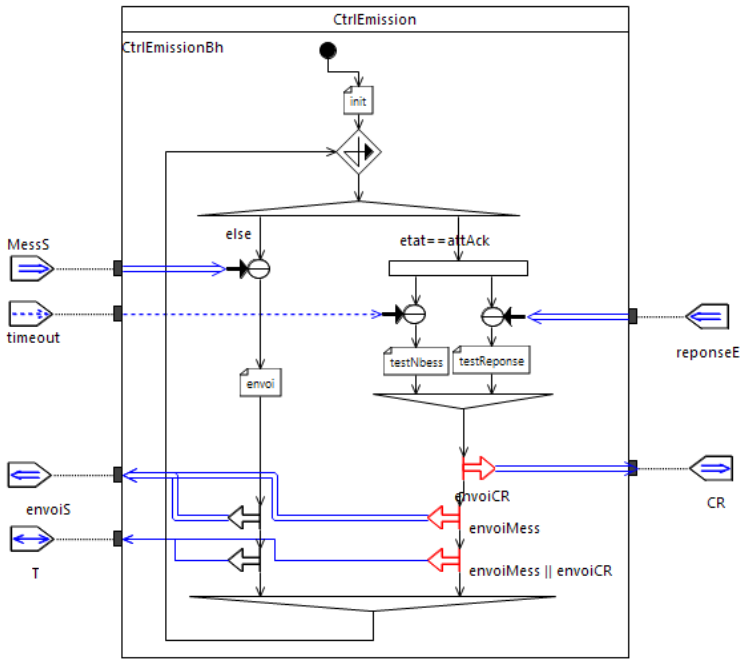


Figure 6 - Activity Chart de la fonction CtrlEmission

Nous utilisons une condition sur l'état pour définir les opérations à effectuer. Si l'état courant est $idle$, le système suivra la branche de gauche, si l'état courant est $attAck$, le système suivra la branche de droite. Dans ce dernier cas, le premier signal qui arrive détermine une nouvelle fois l'opération à effectuer : si $timeout$ arrive avant $reponseE$, on renouvelle l'envoi du message, sinon on accuse réception (si le message reçu est ACK) ou on renouvelle l'envoi (si le message reçu est NACK). Voici les algorithmes utilisés :

```
1 void cfm_op_init()
2 {
3     etat=idle;
4     envoiCR=0;
5     envoiMess=0;
6     T=0;
7     for(int i=0;i<maxchar; i++){
8         buffer[i]=0;
9     }
10 }
```

Figure 7 - Algorithme de la fonction CtrlEmission : init

```
1 void cfm_op_envoi()
2 {
3     //Emission du message
4     for(int i=0;i<maxchar; i++){
5         buffer[i]=MessS[i];
6     }
7     for(int j=0;j<maxchar; j++){
8         envoiS[j]=buffer[j];
9     }
10    T=0;
11    //Mise en attente d'un ACK
12    etat=attAck;
13    nbess=1;
14 }
```

Figure 8 - Algorithme de la fonction CtrlEmission : envoi

```
void cfm_op_testNbess()
{
    cf_trace_report("TIMEOUT");
    envoiCR=0;
    envoiMess=0;
    if(nbess==MaxT){
        cf_trace_report("nbess=%d/%d, Nombre max de tentatives atteint, echec d'envoi", nbess, MaxT);
        envoiCR=1;
        CR=echec;
        etat=idle;
    }else{
        for(int i=0;i<maxchar; i++){
            envoiS[i]=buffer[i];
        }
        envoiMess=1;
        cf_trace_report("nbess=%d/%d, Nouvelle tentative d'envoi du message %s", nbess, MaxT, buffer);
        nbess=nbess+1;
        T=0;
    }
}
```

Figure 9 - Algorithme de la fonction CtrlEmission : testNbess

```
void cfm_op_testReponse()
{
    envoiCR=0;
    envoiMess=0;
    if(reponseE[0]==6){ // Réponse reçue : ACK
        CR=succes;
        envoiCR=1;
        etat=idle;
    }else if(reponseE[0]==21){ // Réponse reçue : NACK
        if(nbess>=MaxT){
            cf_trace_report("Reponse reçue : NACK --> Nombre max de tentatives atteint, echec d'envoi");
            CR=echec;
            envoiCR=1;
            etat=idle;
        }else{
            cf_trace_report("Reponse reçue : NACK --> Nouvelle tentative");
            envoiMess=1;
            nbess=nbess+1;
            T=0;
            for(int i=0;i<maxchar; i++){
                envoiS[i]=buffer[i];
            }
        }

    }
}
```

Figure 10 - Algorithme de la fonction CtrlEmission : testReponse

A.2.2 Fonction Modif_tps

Nous utilisons une fonction permettant d'indiquer l'avancement du temps passé à partir du dernier envoi. Cela permet de générer un évènement $Timeout$ lorsque la réponse attendue ne nous parvient pas à temps (ici, $Timeout$=100ms).

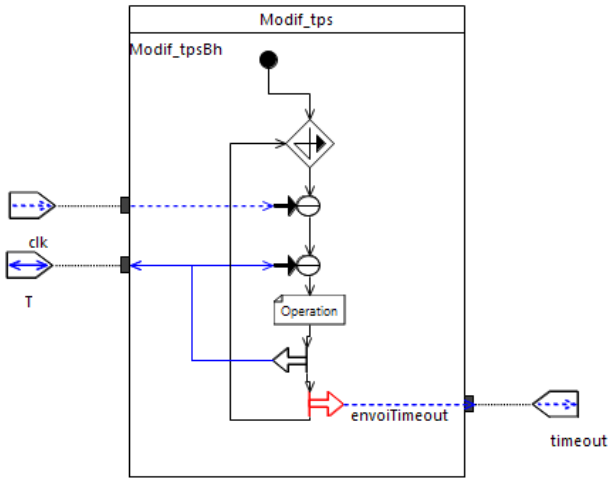


Figure 11 - Activity chart de la fonction Modif_tps

A chaque front d'horloge, T est incrémenté, et lorsqu'il dépasse une valeur T_{max}, un événement $timeout$ est envoyé. Voici l'algorithme utilisé :

```
1 void cfm_op_Operation()
2 {
3     envoiTimeout=0;
4     T=T+PAS;
5     if(T==Tmax){
6         envoiTimeout=1;
7         cf_trace_report("TIMEOUT");
8     }
9 }
```

Figure 12 - Algorithme de la fonction Modif_tps : Operation

A.2.3 Fonction RéceptionLiaison

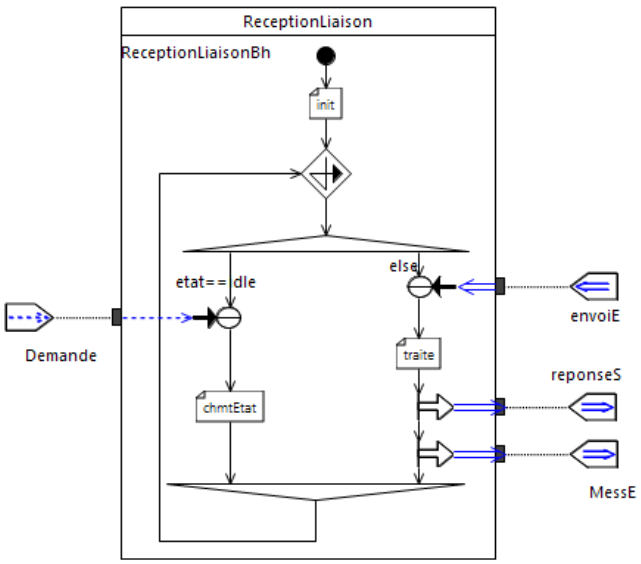


Figure 13 - Activity Chart de la fonction RéceptionLiaison

Nous utilisons une condition sur l'état pour définir les opérations à effectuer. Si l'état courant est $idle$, le système suivra la branche de gauche, si l'état courant est $attmes$, le système suivra la branche de droite. Dans le premier cas, un évènement $demande$ permet un passage à l'état suivant, et dans le second, selon si le message est valide ou non, une réponse ACK ou NACK est émise de même que le message au consommateur. Voici les algorithmes utilisés :

```
1 void cfm_op_chmtEtat()
2 {
3     etat=attmes; // Attente d'un message
4 }
```

Figure 14 - Algorithme de la fonction RéceptionLiaison : chmtEtat

```
1  void cfm_op_traite()
2  {
3      if(envoiE[0]==0){    //Erreur Checksum
4          reponseS[0]=21; // Réponse émise : NACK
5          reponseS[1]=0;
6      }else{
7          reponseS[0]=6; // Réponse émise : ACK
8          reponseS[1]=0;
9          //Transmission du message au consommateur
10         for(int i=0;i<maxchar; i++){
11             MessE[i]=envoiE[i];
12         }
13         etat=idle;
14     }
15
16 }
```

Figure 15 - Algorithme de la fonction RéceptionLiaison : traite

A.3. Couche intermédiaire : Framing/Deframing

A.3.1 Framing

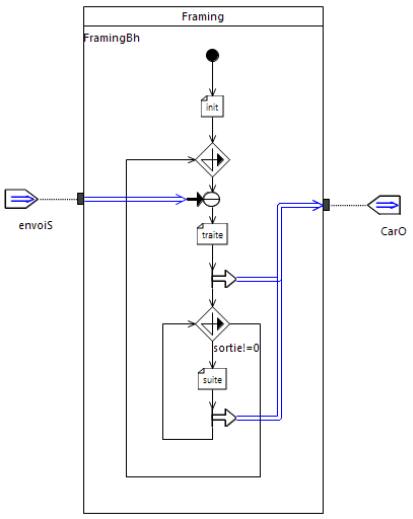


Figure 16 - Activity Chart de la fonction Framing

```
1 void cfm_op_traite()
2 {
3     I=0;
4     CarO=2; //STX
5     sortie=1;
6     checksum=0;
7     somme=2;
8 }
```

Figure 17 - Algorithme de la fonction traite

```
1 void cfm_op_suite()
2 {
3     if(envoiS[I]>3){ //envoi d'un caractère
4         CarO=envoiS[I];
5         somme+=envoiS[I];
6         I++;
7     }else if(envoiS[I]==0 && checksum==0){ //envoi du CheckSum
8         somme+=3;
9         if(somme>255){
10            somme++;
11        }
12        somme=somme%256;
13        CarO=~somme;
14        checksum=1;
15    }else{ // On envoie le caractère de fin de chaine et on sort de la boucle
16        CarO=3;
17        sortie=0;
18    }
19 }
```

Figure 18 - Algorithme de la fonction suite

A.3.2 Deframing

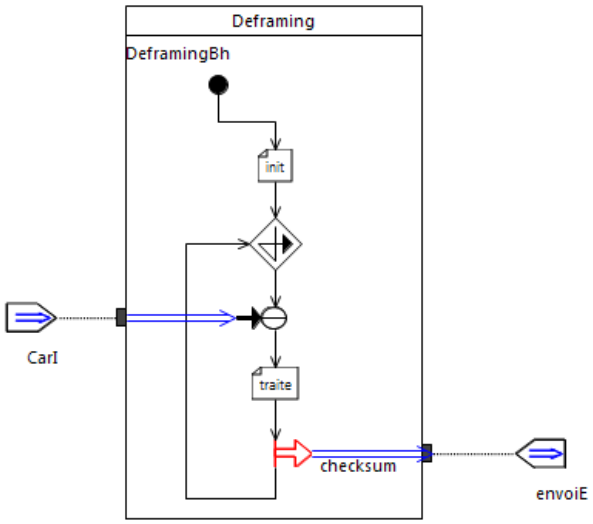


Figure 19 - Activity Chart de la fonction Deframing

```
1 void cfm_op_init()
2 {
3     envoi=0;
4     checksum=0;
5 }
```

Figure 20 - Algorithme de la fonction init

```
1 void cfm_op_traite()
2 {
3     if(CarI==2){ //Start
4         I=0;
5         checksum=0;
6         sommeBuffer=2;
7     }else if(CarI==3){ //Stop
8         Buf[I]=0;
9         sommeBuffer+=3;
10
11        if(sommeBuffer>255){
12            sommeBuffer++;
13        }
14        sommeBuffer=~sommeBuffer;
15        sommeBuffer=sommeBuffer%256;
16
17        if(sommeBuffer==0){ //Les sommes concordent, les données sont valides
18            for(int k=0; k<I-1; k++){
19                envoiE[k]=Buf[k];
20            }
21            envoiE[I-1]=0;
22            checksum=1;
23        }else{ //Message invalidé par la somme
24            envoiE[0]=0;
25            cf_trace_report("Erreur transmission message");
26        }
27    }else{ //Other
28        Buf[I]=CarI;
29        checksum=0;
30        sommeBuffer+=Buf[I];
31        I++;
32    }
33 }
```

Figure 21 - Algorithme de la fonction traite

Le message n'est envoyé que si l'on arrive sur un caractère ETX et que le Checksum est correct (envoi conditionnel).

B. Interface du bip sonore

B.1. Précisions sur le fonctionnement attendu

Le Buzzer consiste en une membrane dont la position dépend de la tension donnée en entrée. Ainsi, pour émettre un son, il convient d'envoyer en entrée un signal dont la fréquence correspond à la fréquence sonore voulue. Dans cette partie nous supposerons un signal carré et non sinusoïdal, dont l'amplitude sera à valeurs fixes à 0V ou 3,3V.

La durée d'un bip sera constante (Tbuzz=40ms), seul le temps de silence entre chaque bip sera modifié en fonction de la distance de l'obstacle de la manière suivante :

Bornes	Bip(s) par seconde	Temps de silence $\frac{1}{nb\ Bip/s} - (Durée\ d'un\ bip)$
0 < d < 25	5 b/s	0,16s
25 < d < 50	4 b/s	0,21s
50 < d < 100	3 b/s	0,29s
100 < d < 150	2 b/s	0,46s
150 < d < 200	1 b/s	0,96s
200 < d	0 b/s	∞

Durée d'un bip : 40ms
Fréquence sonore : 1,25kHz
→ Période : 800µs

⇒

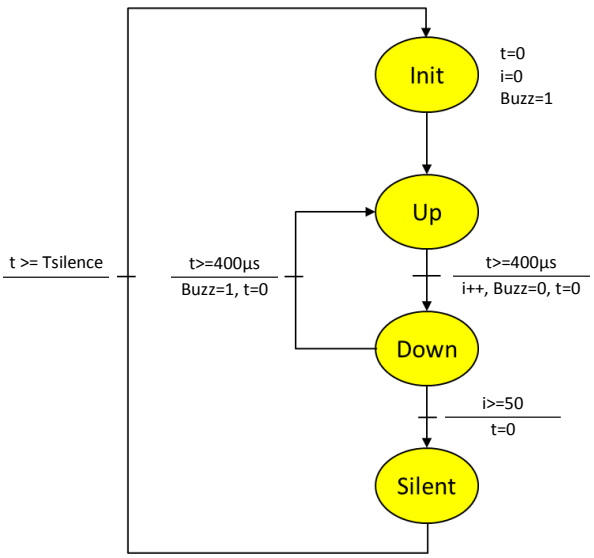
Nombre de périodes à implémenter pour un bip : $\frac{40.10^{-3}}{800.10^{-6}} = 50$ périodes

B.2. Modélisation

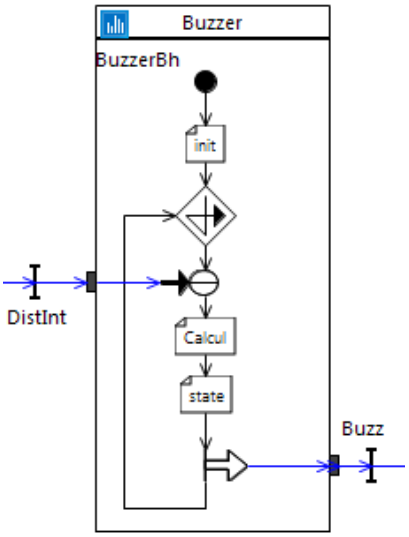
B.2.1 Fonctionnement

Nous optons pour la machine à états suivante, où le passage de l'état Up à $Down$ permet l'oscillation de la membrane créant le bip, et où l'état $Silent$ correspond à l'état d'attente du bip suivant.

La variable $Buzz$ pilote directement la membrane, la variable interne *i* compte le nombre d'oscillations de la membrane, la variable t est naturellement incrémentée en fonction du temps, et enfin, la variable $T_{silence}$ est calculée précédemment par le Gestionnaire d'Interfaces en fonction de la distance de l'obstacle (selon tableau précédent).



B.2.2 Modélisation sous Cofluent



Nous utilisons une fonction échantillonnée à 400µs dont la boucle principale reçoit la distance au format *Integer*, calcule la valeur de $T_{silence}$ et entre dans le switch..case de la machine à état décrite précédemment. En sortie nous obtenons le signal $Buzz$ permettant de piloter la membrane du buzzer.

B.2.3 Résultat de simulation sous Cofluent

Nous effectuons une simulation sur 6 secondes en considérant que la distance de l'obstacle est décroissante entre 231 cm et 32 cm.

```
[...]
<REPORT>:0 us : RadarDeRecul/init> Init_RadarDeRecul
<REPORT>:1 us : LevierDeVitesse/MarcheArriere> posFalse
<REPORT>:8002 us : LevierDeVitesse/RemarcheAvant> posTrue
<REPORT>:21629 us : CtrlEmission/envoi> Nouveau message emis : 231
<REPORT>:23840 us : Recuperation/StringToInt> Distance decode 231
<REPORT>:524292 us : CtrlEmission/envoi> Nouveau message emis : 208
<REPORT>:526501 us : Recuperation/StringToInt> Distance decode 208
<REPORT>:1027004 us : CtrlEmission/envoi> Nouveau message emis : 186
<REPORT>:1029217 us : Recuperation/StringToInt> Distance decode 186
<REPORT>:1029222 us : Buzzer/state> BUZZ !!!
<REPORT>:1529729 us : CtrlEmission/envoi> Nouveau message emis : 164
<REPORT>:1531941 us : Recuperation/StringToInt> Distance decode 164
<REPORT>:2029222 us : Buzzer/state> BUZZ !!!
<REPORT>:2032442 us : CtrlEmission/envoi> Nouveau message emis : 142
<REPORT>:2034653 us : Recuperation/StringToInt> Distance decode 142
<REPORT>:2529222 us : Buzzer/state> BUZZ !!!
<REPORT>:2535167 us : CtrlEmission/envoi> Nouveau message emis : 120
<REPORT>:2537377 us : Recuperation/StringToInt> Distance decode 120
<REPORT>:3029222 us : Buzzer/state> BUZZ !!!
<REPORT>:3037879 us : CtrlEmission/envoi> Nouveau message emis : 098
<REPORT>:3040089 us : Recuperation/StringToInt> Distance decode 98
<REPORT>:3362222 us : Buzzer/state> BUZZ !!!
<REPORT>:3540604 us : CtrlEmission/envoi> Nouveau message emis : 076
<REPORT>:3542817 us : Recuperation/StringToInt> Distance decode 76
<REPORT>:3695222 us : Buzzer/state> BUZZ !!!
<REPORT>:4028222 us : Buzzer/state> BUZZ !!!
<REPORT>:4043317 us : CtrlEmission/envoi> Nouveau message emis : 054
<REPORT>:4045529 us : Recuperation/StringToInt> Distance decode 54
<REPORT>:4361222 us : Buzzer/state> BUZZ !!!
<REPORT>:4546042 us : CtrlEmission/envoi> Nouveau message emis : 032
<REPORT>:4548253 us : Recuperation/StringToInt> Distance decode 32
<REPORT>:4611222 us : Buzzer/state> BUZZ !!!
<REPORT>:4861222 us : Buzzer/state> BUZZ !!!
```

```
<REPORT>:5111222 us : Buzzer/state> BUZZ !!!
<REPORT>:5361222 us : Buzzer/state> BUZZ !!!
<REPORT>:5611222 us : Buzzer/state> BUZZ !!!
<REPORT>:5861222 us : Buzzer/state> BUZZ !!!
<INFO>:6000000 us:/> Reached the specified simulation duration:
<INFO>:6000000 us:/>   current simulated time:     6000000 us
<INFO>:6000000 us:/>   host system clock duration: 218 s
<INFO>:6000000 us:/>   transaction count:          69882567
<INFO>:6000000 us:/>   average speed:              3.2056e+005 transactions/s
[info]  Shutdown OK
```

Distance reçue : 231 cm puis 208 cm

⇒ 200 < distance

⇒ Tsilence infini

⇒ pas de buzz

Distance reçue : 186 cm puis 164 cm

⇒ 150 < distance < 200

⇒ Tsilence = 960 ms (1 bip/s)

⇒ buzz à t=1 029 ms, puis à $t+T_{buzz}+T_{silence}$ = 2 029 ms

Distance reçue : 142 cm puis 120 cm

⇒ 100 < distance < 150

⇒ Tsilence = 460 ms (2 bips/s)

⇒ buzz à t=2 029 ms, puis à $t+T_{buzz}+T_{silence}$ = 2 529 ms, 3 029 ms

Distance reçue : 98 cm puis 76 cm puis 54 cm

⇒ 50 < distance < 100

⇒ Tsilence = 290 ms (3 bips/s)

⇒ buzz à t=3 029 ms, puis à $t+T_{buzz}+T_{silence}$ = 3 362 ms, 3 695 ms, 4 028 ms, 4 361 ms

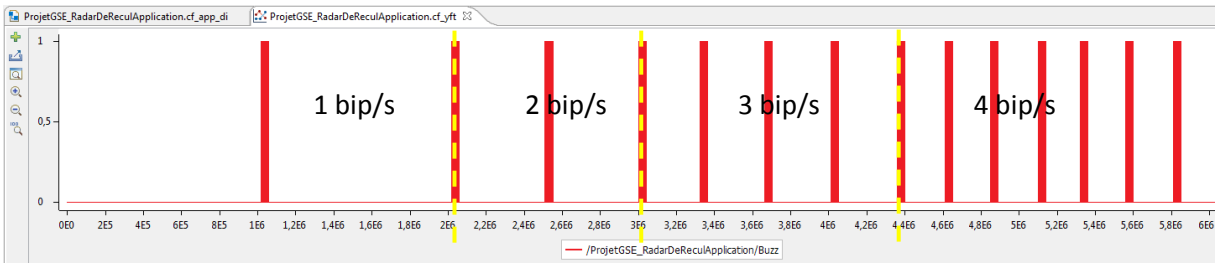
Distance reçue : 32 cm

⇒ 25 < distance < 50

⇒ Tsilence = 210 ms (4 bips/s)

⇒ buzz à t=4 361 ms, puis à $t+T_{buzz}+T_{silence}$ = 4 611 ms, 4 861 ms, 5 111 ms, 5 361 ms…

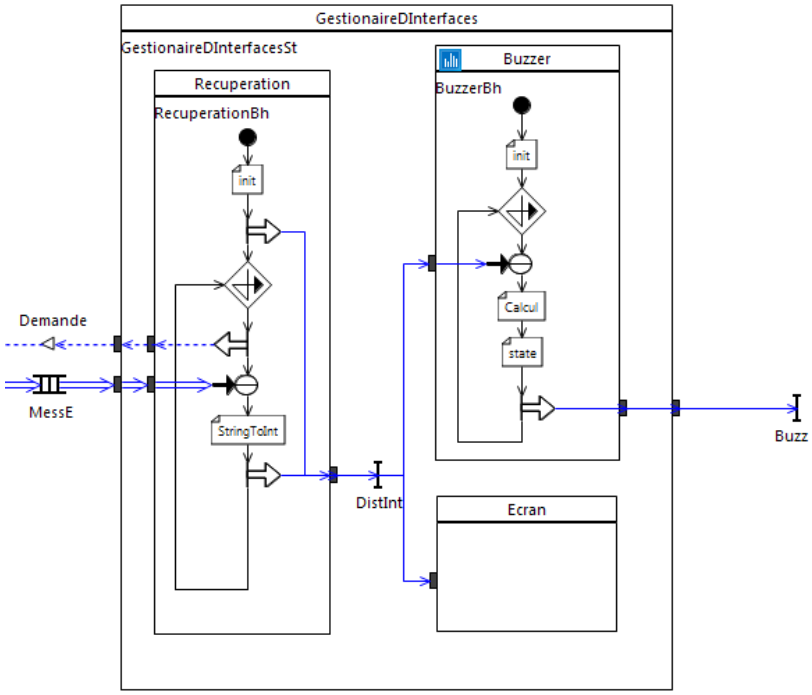
L'intervalle de temps entre 2 bips correspond bien à ce qui est souhaité.



C. Gestionnaire d'interfaces

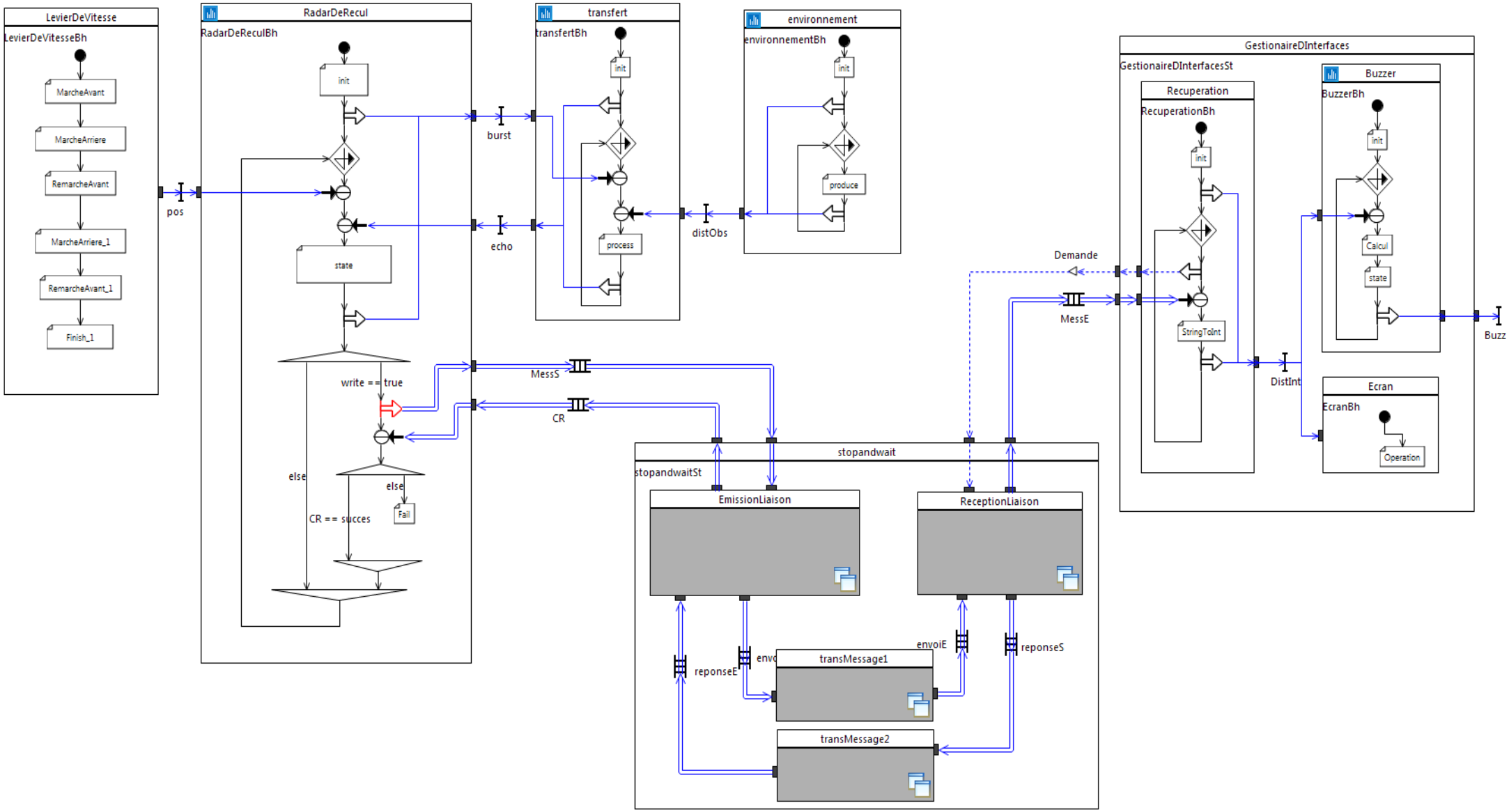
La gestion du Buzzer est intégrée à un Gestionnaire d'Interfaces (GI), permettant le contrôle des périphériques en fonction de la distance reçue par la transmission orientée caractères.

Une opération capitale effectuée par le GI est la conversion des données de distance en format entier, afin d'être utilisable par l'ensemble des fonctions. Le Buzzer par exemple, en a besoin pour la manipulation de la membrane sonore. L'écran en fera également usage afin d'adapter l'interface graphique en intégrant un indicateur proportionnel à la distance sur l'affichage.



D. Modèle complet du système

Sur la page suivante, nous rappelons le modèle Cofluent complet, qui intègre les différents éléments présentés tout au long de ce rapport.


LevierDeVitesse
LevierDeVitesseBh
MarcheAvant
MarcheArriere
RemarcheAvant
MarcheArriere_1
RemarcheAvant_1
Finish_1
pos
RadarDeRecul
RadarDeReculBh
init
state
write == true
else
else
CR == succes
Fail
burst
echo
transfert
transfertBh
init
process
distObs
environnement
environnementBh
init
produce
MessS
CR
stopandwait
stopandwaitSt
EmissionLiaison
ReceptionLiaison
transMessage1
transMessage2
reponseE
envoiE
reponseS
Demande
MessE
GestionaireDInterfaces
GestionaireDInterfacesSt
Recuperation
RecuperationBh
init
StringToInt
DistInt
Buzzer
BuzzerBh
init
Calcul
state
Buzz
Ecran
EcranBh
Operation

Conclusion de la partie 3

Nous avons pu préciser, dans cette partie, le protocole de communication que nous allons utiliser entre nos 2 plateformes. Nous avons précisé chaque étape du protocole, notamment la couche de liaison, la couche framing/deframing et la couche de transmission caractère par caractère.

Nous avons également pu développer le fonctionnement attendu du Buzzer et le modéliser sous Cofluent. Enfin, nous l'avons intégré dans la modélisation du Gestionnaire d'Interfaces qui est à même de piloter les périphériques en fonction de la distance reçue en entrée.

Notre modèle fonctionne conformément aux exigences, nous allons pouvoir nous attacher à la phase de prototypage.

Partie 4 : Prototypage

Objectif : L'objet de cette quatrième partie du projet est d'implémenter sur carte STMicroelectronics les modèles que nous avons définis. Nous allons donc appliquer les algorithmes définis précédemment au programme principal de chaque plateforme.

Nous utiliserons deux cartes STMicroelectronics : la première est une carte BCMSTM32F400, la seconde est une carte STM32Nucleo-L152RET6.

A. Implémentation du protocole de communication

Le protocole de communication comprend la couche de liaison et la couche de framing/deframing. La couche caractère étant prise en charge par le matériel, nous ne nous chargerons pas de son implémentation niveau bit, mais simplement du contrôle d'émission/réception orienté caractère.

A.1. Liaison

La fonction (simplifiée ci-dessous) effectue l'envoi puis la réception du message. La fonction *receive* retourne un code d'exécution dépendant de la validité de la trame reçue (couche inférieure) : si une erreur est détectée ou si le temps de réception dépasse le Timeout imposé, ou encore si un message NACK est reçu, alors l'envoi est renouvelé jusqu'à 3 fois.

```
void liaison(message){
    essai=0;
    send(message);
    err=receive(trame);

    while((controle(trame)==0 || err==1) && essai<3){
        send(message);
        err=receive(trame);
        essai++;
    }
}
```

A.2. Framing/Deframing

A.2.1 En émission :

La couche de framing permet de mettre en forme la trame à émettre, en intégrant notamment un caractère de début de trame, de fin de trame et un Checksum. Une fois cette opération effectuée, la trame est envoyée caractère par caractère par la fonction *USART1_puts*.

```
void send(char *data){
        char trame[52]="";
        format(data, trame);
        USART1 puts(trame);
}
```

La fonction de mise en forme de la trame insère le caractère STX (0x02) en début de chaine, puis ajoute les caractères du message à envoyer, avant de calculer la somme de toutes les valeurs associées aux caractères de la chaine selon la formule de calcul d'un Checksum vue l'année précédente :

Si la somme des caractères (STX et ETX inclus) dépasse 255, le checksum est incrémenté d'une unité est un modulo 256 lui est appliqué. Sa valeur est finalement inversée bit à bit avant d'être intégrée à la chaine de caractères.

Enfin, le caractère ETX (0x03) est disposé en fin de trame, ainsi qu'un caractère null de sécurité.

```
void format(char *msg, char *msg_formated){
    int somme=5;
    int i=0;
    msg_formated[0]=2;
    while(msg[i]!=0){
        somme= somme + msg[i];
        msg_formated[i+1]=msg[i];
        i++;
    }
    if(somme>255){
        somme++;
        somme=somme%256;
    }
    msg_formated[i+1]=~somme;
    msg_formated[i+2]=3;
    msg_formated[i+3]=0;
}
```

A.2.2 En réception :

La réception est également effectuée caractère par caractère, du caractère STX jusqu'au ETX, en vérifiant le Checksum à postériori. Si une erreur est détectée, ou si le Timeout arrive à son terme avant la fin de la réception (ici, Timeout=100ms), la valeur 1 est retourné, alors que si la trame est reconstruite sans erreur dans le temps imparti, la valeur 0 est retournée.

```
int receive(char *frame){
    int I;
    unsigned int sommeBuffer;
    char r;
    Timer timeout;
    timeout.start();
    while(1){
        if(timeout.read_ms() > 100) return 1;
        if(carte.readable()){
            r = carte.getc();

            switch(r){
                case 2: //Start
                    I=0;
                    sommeBuffer=2;
                    break;

                case 3: //Stop
                    frame[I]=0;
                    sommeBuffer+=3;

                    if(sommeBuffer>255){
                        sommeBuffer++;
                    }
                    sommeBuffer=~sommeBuffer;
                    sommeBuffer=sommeBuffer%256;

                    if(sommeBuffer==0){ //Les sommes concordent, les données sont valides
                        frame[I-1]=0;
```

```
                        return 0;
                    }else{ //Message invalidé par la somme
                        frame[0]=0;
                        return 1;
                    }

                default :
                    frame[I]=r;
                    sommeBuffer+=r;
                    I++;
                    break;
            }
        }
    }
}
```

A.3. Transmission de caractères

Nous allons ici transmettre les caractères un à un sur le canal, mais sans effectuer de décomposition bit à bit, ni calcul de parité. Nous utilisons la fonction *USART1_puts* en émission, et la fonction *getc* en réception.

```
void USART1_puts(char *text){
    while(*text != '\0'){

        if(carte.writeable()){
            carte.putc(*text);
            text++;
        }
    }
}
```

En émission comme en réception, nous devons vérifier l'état des registres avant d'effectuer toute opération. Par exemple, ci-dessus nous veillons à ce que la carte puisse envoyer des données (FIFO d'émission vide) avant d'appeler la fonction *putc()*.

Pour ces opérations, nous devons nous adapter à la plateforme utilisée. La fonction ci-dessus est un exemple d'utilisation pour la carte Nucleo, mais la syntaxe sera différente et sensiblement plus complexe pour la STM32F400.

A.4. Test de la transmission

Pendant la phase de test nous simulons un échec de Checksum en insérant une erreur dans l'algorithme, puis nous simulons en condition normale d'utilisation.

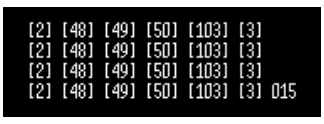


Figure 22 - Erreur volontaire dans la transmission

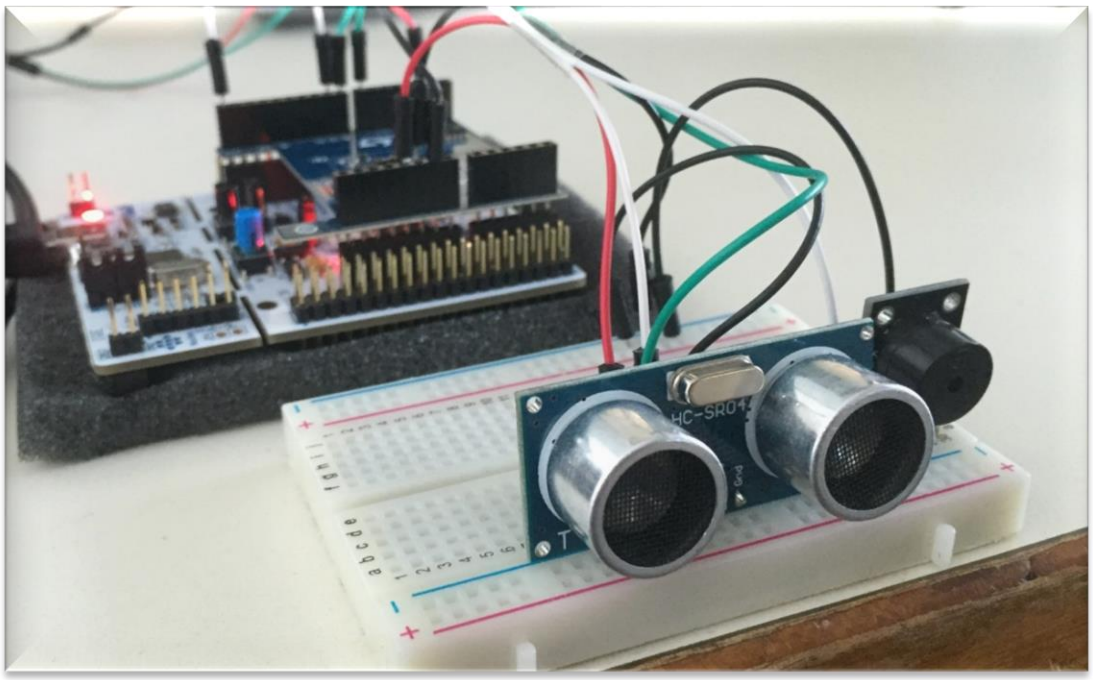
```
[2] [48] [55] [54] [93] [3] 096
[2] [48] [57] [54] [91] [3] 096
[2] [48] [57] [54] [91] [3] 078
[2] [48] [55] [56] [91] [3] 094
[2] [48] [57] [52] [93] [3] 108
[2] [49] [48] [56] [97] [3] 096
[2] [48] [57] [54] [91] [3] 080
[2] [48] [56] [48] [98] [3] 097
```

Figure 23 - Transmission en condition normale d'utilisation

On observe bien une retransmission jusqu'à 3 fois du message lorsqu'une erreur de Checksum est détectée, et une transmission fluide en situation normale.

B. Implémentation de l'émetteur/récepteur, du buzzer et du levier de vitesse

La gestion de l'émetteur / récepteur, de la transmission des données, du Buzzer et de la position du levier de vitesse sera implémentée sur la carte Nucleo-L152RET6 sous forme d'OS temps réel. Cela permet de n'avoir aucune attente active (bloquante) et pouvoir ainsi lancer des tâches à des fréquences différentes.



B.1. Emetteur / Récepteur

Le capteur de distance utilisé pour cette première phase de prototypage sera de type HCSR-04. Celui-ci fonctionne sur le principe suivant : un signal de 10µs TTL en entrée agit comme signal de départ pour la prise de mesure. Le capteur émet alors une onde ultrason et attend son retour. Il convertit ensuite le temps de réponse en signal de sortie TTL de longueur proportionnelle (1m ↔ 5 800µs).

Table1 - Rapport entre la distance mesurée par le capteur et la durée de son signal de sortie

	Documentation		Nécessaire au projet	
	Référence	Résolution	Min	Max
Distance mesurée	100 cm	0,3 cm	17 cm	200 cm
Durée du signal de sortie	5 800 µs	17 µs	986 µs	11 600 µs

Nous implémentons alors une fonction permettant de générer le signal TTL de 10µs (Trig), et une fonction sensible aux fronts montant et descendant du signal de sortie (Echo).

void send_burst()	Fonction qui génère le burst pour la mesure de distance, et remet à 0 le Timer associé à la mesure
void echo_rise()	Démarre le Timer permettant mesurant la durée du signal de sortie du capteur (durée proportionnelle à la distance)
void echo_fall()	Stoppe le timer, récupére la distance en cm et modifie le silence entre deux bip du Buzzer (voir partie suivante)

Voici le détail des fonctions correspondantes à intégrer dans notre programme sur la carte Nucleo.

```
/**Fonction qui génère le burst pour la mesure de distance.**/
//But : Générer un burst et reset le timer de distance.
//Donnée :  Le timer distance_time est un timer global.
//          trig est un sortie digital.
//
// TRIG => ________/              \__....
//          2 µs         10 µs
void send_burst(){
    distance_time.reset();
    trig = 0;
    wait_us(2);
    trig = 1;
    wait_us(10);
    trig = 0;
}
/********Routine d'interruption sur front montant du signal echo.*********/
//But : Lancer le timer.
//
//Donnée :  Le timer distance_time est un timer global.
//                     ...
// ECHO => ...    /
//                ^  Start timer.
void echo_rise(){
    distance_time.start();
}
/***********Routine d'interruption sur front descendant du signal echo.******/
//But : Stopper le timer, récupérer la distance en cm
//      et modifier le silence entre deux bip du buzzer
//
//Données:  Le timer distance_time est un timer global.
//          L'entier distance est une variable global.
//          Le buzzer buzzo est un objet global.
```

```
//                    _____________________________________
// ECHO => _________/      Distant = Temps / 29/2          \__.... Stop timer.
//                                                          ^
//
void echo_fall(){
    distance_time.stop();  //Stoppe le timer.

    distance = distance_time.read_us() /29 / 2; //Convertit le temps (µs) en distance (cm).
    distance = min(distance,230);   //Définit la valeur max de la distance à 230.
    distance = max(distance,0);     //Définit la valeur min de la distance à 0.

    buzzo.changeMode(distance);     //Mise à jour de l'intervalle entre chaque bip
}
```

B.2. Bip sonore, ou buzzer

L'état du Buzzer est mis à jour à une fréquence de 400µs, le micro-contrôleur a donc le temps d'effectuer d'autres opérations dans l'intervalle disponible. Le déclenchement est opéré à l'aide d'un timeout régulier dont la précision est de 5% ce qui conviendra pour notre périphérique.

Une classe Buzzer est utilisée pour la gestion de celui-ci. La méthode *changeMode()* modifie la valeur de *Tsilence* (correspond à la fonction *Calcul* sous Cofluent), et la méthode *run()* permet l'entrée dans la machine à états (correspond à la fonction *state* sous Cofluent).

```
class Buzzer{
  public :
    Buzzer(){
        TsilenceMs=100;
        etat = BipInit;
        t.reset();
        i=0;
    }
    int TsilenceMs;

    typedef enum {BipInit, BipUp, BipDown, Silent} defetat;
    defetat etat;
    Timer t;
    int i;

    void changeMode(int value);
    void run();
};
```

Voici notamment les détails des méthodes *changeMode()* et *run().*

```
//Modifie la valeur du temps de silence entre 2 bips en fonction de la distance
//(Note : Tbuzz = 40ms)
void Buzzer::changeMode(int value){
    if(value<=25){                      //5Bips/s
        TsilenceMs=(1000/5)-Tbuzz;
    }else if(value>25 && value<=50){    //4Bips/s
        TsilenceMs=(1000/4)-Tbuzz;
    }else if(value>50 && value<=100){   //3Bips/s
        TsilenceMs=(1000/3)-Tbuzz;
    }else if(value>100 && value<=150){  //2Bips/s
        TsilenceMs=(1000/2)-Tbuzz;
    }else if(value>150 && value<=200){  //1Bips/s
        TsilenceMs=(1000/1)-Tbuzz;
    }else{                              //0Bips/s
        TsilenceMs=1000000000;
    }
}

// Machine à états : effectue les opérations nécessaires au fonctionnement du buzzer
```

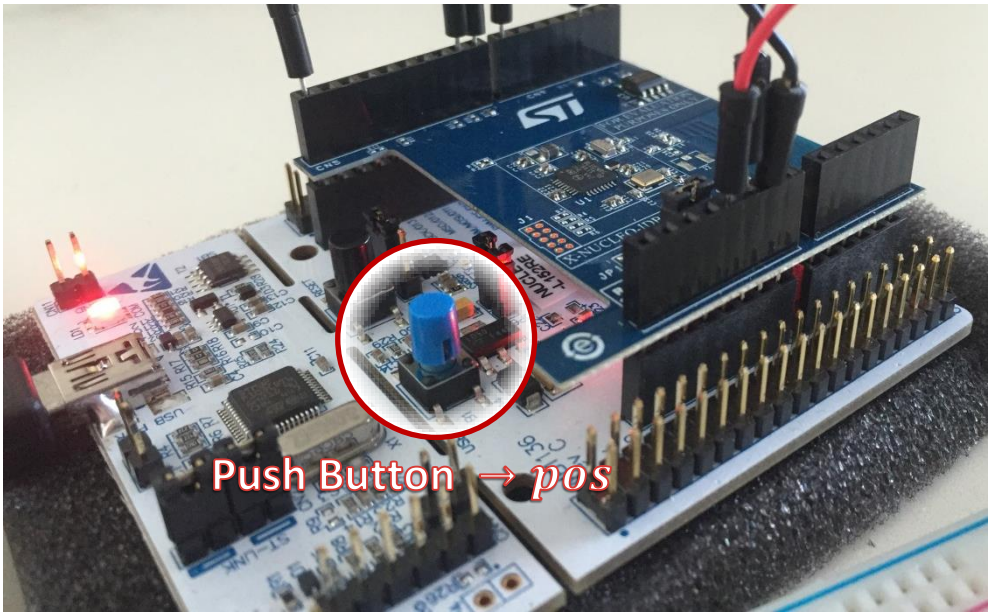
```
void Buzzer::run(){
    switch(etat){

        case BipInit :
            buzz=0;
            i=0;
            etat = BipUp;
            t.start();
            break;

        case BipUp :
            if(t.read_us()>=400 && i<50){
                buzz=1;
                i++;
                etat = BipDown;
                t.reset();
                t.start();
            }
            break;

        case BipDown :
            if(t.read_us()>=400 && i<50){
                buzz=0;
                i++;
                etat = BipUp;
                t.reset();
                t.start();
            }
            if(i>=50){
                etat = Silent;
                t.reset();
                t.start();
            }
            break;

        case Silent :
            if(t.read_ms()>=TsilenceMs){
                etat=BipInit;
            }
            break;

        default :
            etat = BipInit;
            break;
    }
}
```

B.3. Position du levier de vitesse

Dans le but de limiter la consommation lorsque la voiture est en marche avant, nous rendons le système sensible aux fronts montant et descendant du signal *pos* (mise en route de la marche arrière, mise en route de la marche avant).

Ainsi, lorsque *pos* passe à 0, les routines de mesure de distance et de transmission des données (prises en charge par la carte Nucleo sous forme de RTOS) sont misent en route, puis seront arrêtées lorsque *pos* reviendra à 1.

Physiquement, nous utiliserons un bouton poussoir disponible sur la carte Nucleo : pos vaut 0 lorsque l'on enfonce le bouton, puis passe à 1 lorsqu'on le relâche.




```
/********Routine d'interruption sur front descendant du signal pos.*********/
//But : Lancer le programme (Marche arrière enclenchée)
void position fall(){
        echo.rise(&echo rise);   //Routine sur interruption de echo en front montant.
        echo.fall(&echo_fall);   //Routine sur interruption de echo en front descendant.
        mesure_time.attach(&send_burst, 0.5);       //Lance send_burst toutes les 500ms.
        wait(0.1);                                  //Décalage entre send et transmit.
        send_time.attach(&transmit_data, 0.5);      //Lance transmist_data toutes les 500ms.
        wait(0.1);                                  //Décalage entre transmit et buzz.
        bip.attach us(&buzzo, &Buzzer::run, 400);   //Lance l'actualisation du buzzer toutes
les 400µs ( F = 2.5 kHz ).
}

/********Routine d'interruption sur front montant du signal pos.*********/
//But : Stopper les routines en cours (Marche avant enclenchée)
void position rise(){

        echo.rise(NULL);   //Routine NULL sur interruption de echo en front montant.
        echo.fall(NULL);   //Routine NULL sur interruption de echo en front descendant.
        mesure time.detach();    //Stoppe la routine send burst.
        send_time.detach();      //Stoppe la routine transmist_data.
        bip.detach();            //Stoppe la routine l'actualisation du buzzer.
}

int main() {
        position.rise(&position rise); //Routine sur interruption de pos en front montant.
        position.fall(&position fall); //Routine sur interruption de pos en front descendant.
    while(1);
}
```

C. Implémentation du gestionnaire d'interfaces

L'implémentation de l'écran est effectuée sur la carte STM32F400 en langage C pour des raisons de compatibilité. Nous n'utiliserons donc pas de classe, comme cela a pu être fait sur la carte Nucleo.

C.1. Fonction implémentées

C.1.1 Fonctions de communication

`void USART1_Initialize(void)`	Initialise l'UART
`void GPIO_Initialize(void)`	Initialise le GPIO
`int USART1_transmitter_empty(void)`	Teste l'état du canal d'émission
`int USART1_receiver_ready(void)`	Teste l'état du canal de réception
`void USART1_puts(char *s)`	Emet une chaine de caractères.
`void format(char *msg, char *msg_formated)`	Formate la trame de données à émettre
`void send(char *data)`	Formate et envoie une trame de données
`bool receive(char *frame)`	Reçoit une chaine de caractères

C.1.2 Fonction de conversion

`int trametoint(char trame[3])`	Convertit un tableau de char en int

C.1.3 Fonctions liées au fonctionnement de l'écran

`bool receive(char *frame)`	Reçoit une chaine de caractères
`void makeVecteur(int distance, char vecteur[60])`	Vecteur graphique symbolisant la distance restante
`void draw_voiture(int x0 , int y0)`	Affiche la voiture à l'écran
`void clean_voiture(int x0 , int y0)`	Efface la voiture

C.2. Programme principal : Main

Les premières instructions du programme principal, avant l'entrée dans la boucle infinie, ont pour objectif d'initialiser le GPIO puis l'écran LCD embarqué, avant de mettre en place les éléments graphiques de base (Fond d'écran, couleur et taille de la police, titre, label, illustration…).

Nous ne détaillerons pas le contenu des fonctions d'initialisation et de manipulation du GPIO, de l'USART et de l'écran, dont la structure interne est propre à la plateforme BCMSTM32F400. Le code complet est disponible en Annexe.

Après la phase d'initialisation, le programme est prêt à recevoir les données de distance depuis la carte Nucleo. Si la réception est valide, un acquittement est renvoyé, la distance reçue en format caractères est convertie en entier, et l'affichage à l'écran est mis à jour de manière appropriée. Si la

réception n'est pas valide, un message NACK est renvoyé, et une notification d'erreur est affichée à l'écran.

```
int main(){
    char vecteur[60] = "                                                                \0";
    char data[3];
    int dist = 0;
    int i;

    GLCD_Initialize();          //Initialisation de l'écran LCD.
    GLCD_WindowMax();           //Definie la taille maximum de l'écran.
    GLCD_Clear(White);          //Efface l'écran (Affichage Blanc.
    GLCD_SetBackColor(White);   //Definie la couleur de fond à blanc.


    draw_voiture(2,110); //Dessine la voiture à la position x=2 , y=110. (Valeur en pixel).

    GLCD_SetTextColor(Red);                         //Change la couleur d'écriture en rouge.
    GLCD_DisplayString(1,3,1,"Radar de recul");
    GLCD_SetTextColor(Black);                       //Change la couleur d'écriture en noir.
    GLCD_DisplayString(3,1,1,"Distance :     cm");
    USART1_Initialize();    //Initialisation de l'UART
    GPIO_Initialize();      //Initialisation des GPIOs

    while(1){
        if(receive(data)){              //Attente de la réception d'un message.
            send("\x06");               //Si la réception est valide, envoie un acquittement.
            dist = trametoint(data);    //Convertit la trame en entier.
            GLCD_SetTextColor(Black);   //Change la couleur d'écriture en noire.
            GLCD_DisplayString(3,12,1,data);    //Ecrit la distance (3 caractères) à la position x = 3 , y
= 12 (Valeur en caratére) et avec la police 1 (Grande).
            data[0] = 0;
            makeVecteur(dist,vecteur);  //Création du vecteur symbolisant la distance.
            GLCD_DisplayString(17,11,0,vecteur); //Ecrit le vecteur de distance sur 4 ligne.
            GLCD_DisplayString(18,11,0,vecteur); //Taille des caractères 0
            GLCD_DisplayString(19,11,0,vecteur);
            GLCD_DisplayString(20,11,0,vecteur);

            if(dist <= 17){         //Si la distance <= 17 cm, affiche "Attention choc !"
                GLCD_SetTextColor(Red);
                GLCD_DisplayString(8,2,1,"Attention choc !");
                GLCD_SetTextColor(Black);
            }
            else if( dist >= 200 ){ //Si la distance >= 200 cm, affiche "Out of range"
                GLCD_SetTextColor(Red);
                GLCD_DisplayString(8,2,1,"  Out of range ");
                GLCD_SetTextColor(Black);
            }else{    //Sinon affiche rien.
                GLCD_DisplayString(8,2,1,"                ");
            }

        }
        else{
            send("\x21");   //Si la réception n'est pas valide renvoie un non acquittement.
            GLCD_SetTextColor(Red);
            GLCD_DisplayString(8,2,1,"  Data Error ! ");    //Affiche "Data Error !"
            GLCD_SetTextColor(Black);
        }
    }
}
```

D. Implémentation générale

Conclusion de la partie 4

Dans cette partie, nous nous sommes confrontés au fonctionnement spécifique des cartes STMicroelectronics, notamment l'implémentation de timer, de timeout, l'utilisation du GPIO, de l'écran LCD, etc. Nous avons donc adapté les algorithmes de la partie précédente pour opérer la communication souhaitée entre les cartes, et implémenter correctement le Buzzer et l'écran LCD.

En pratique, seul l'écran sera réellement implémenté sur la carte STM32F400, le Buzzer sera quant à lui géré par la carte Nucleo (pour des raisons d'accessibilité aux pins du GPIO).

Notre système est pleinement fonctionnel et répond au cahier des charges du client.

Conclusion générale

Ce projet de conception d'un radar de recul est une introduction au travail auquel est confronté un ingénieur en électronique, mais dans une dimension adaptée au calendrier universitaire.

Nous avons pu notamment nous confronter à la spécification du système en construisant une première base de données structurée à partir de son cahier des charge. Nous avons ensuite proposé une solution d'implémentation du modèle décrit lors de la première étape en termes de fonctions et de relations, et simulé son fonctionnement sous Cofluent. Nous avons également approfondi le protocole de communication interne entre les plateformes du système et la gestion des interfaces en modélisant notamment leur fonctionnement et leur gestion. Enfin, nous avons implémenté les modèles que nous avons défini et leurs algorithmes sur plateforme STMicroelectronics.

Cette expérience nous a permis d'enrichir notre façon de travailler en équipe, de réagir avec justesse face à des problèmes techniques, de nous libérer de situations de blocage, et de mener à bien notre projet dans les temps impartis. Notre radar est maintenant fonctionnel, et pourrait constituer un support dans le cadre d'un travail de plus grande dimension en explorant davantage son industrialisation, son intégration en environnement réel.

Annexes

A. Radar : STM32Nucleo-L152RET6

A.1. Aperçu

Structure de fichiers :

- UART_Radar.h
- UART_Radar.cpp
- buzzer.h
- main.cpp

A.2. Fichiers de code

A.2.1 UART_Radar.h

```
#ifndef UART_RADAR_H
#define UART_RADAR_H

//Communication UART : envoi
void USART1_puts(char *text);

// Formattage de la trame STX, CRC, ETX
void format(char *msg, char *msg formated);

//Formatage de la trame et envoi via UART
void send(char *data);

//Fonction de réception d'un message
//Si le Checksum est invalide ou si la réception dépasse le Timeout, une erreur est renvoyée
int receive(char *frame);

//Vérification de l'Acknowledge
int controle(char *frame);

#endif
```

A.2.2 UART_Radar.cpp

```
#include "UART_Radar.h"
#include "mbed.h"
#include <string.h>

Serial carte(PA_9, PA_10); //emission --- tx, rx
Serial pc(USBTX, USBRX); //réception --- tx, rx

//Communication UART : envoi
void USART1_puts(char *text){
    while(*text != '\0'){

        if(carte.writeable()){
            carte.putc(*text);
            text++;
        }

    }
}
```

```
// Formattage de la trame STX, CRC, ETX
void format(char *msg, char *msg_formated){
    int somme=5;
    int i=0;
    msg formated[0]=2;
    printf("\n\r");
    while(msg[i]!=0){
        somme= somme + msg[i];
        msg_formated[i+1]=msg[i];
        i++;
    }
    if(somme>255){
            somme++;
    }
    somme=somme%256;
    msg formated[i+1]=~somme;
    msg formated[i+2]=3;
    msg_formated[i+3]=0;
}

//Formatage de la trame et envoi via UART
void send(char *data){
        char trame[52]="";
        format(data, trame);
        USART1_puts(trame);
}

//Fonction de réception d'un message
//Si le Checksum est invalide ou si la réception dépasse le Timeout, une erreur est renvoyée
int receive(char *frame){
    int I;
    unsigned int sommeBuffer;
    char r;
    Timer timeout;
    timeout.start();
    while(1){
        if(timeout.read_ms() > 100) return 1;
        if(carte.readable()){
            r = carte.getc();

        switch(r){
            case 2: //Start
                I=0;
                sommeBuffer=2;
                break;

            case 3: //Stop
                frame[I]=0;
                sommeBuffer+=3;

                if(sommeBuffer>255){
                    sommeBuffer++;
                }
                sommeBuffer=~sommeBuffer;
                sommeBuffer=sommeBuffer%256;

                if(sommeBuffer==0){ //Les sommes concordent, les données sont valides
                    frame[I-1]=0;
                    return 0;
                }else{ //Message invalidé par la somme
                    frame[0]=0;
                    return 1;
                }

            default :
                frame[I]=r;
                sommeBuffer+=r;
                I++;
                break;
            }
        }
    }
}

//Vérification de l'Acknowledge
int controle(char *frame){
    if (*frame!=6) return 0;
```

```
    else return 1;
}
```

A.2.3 buzzer.h

```
#ifndef BUZZER_H
#define BUZZER H

#include "mbed.h"

DigitalOut buzz(PB_8);


class Buzzer{
    public :
    Buzzer();
    int TsilenceMs;

    typedef enum {BipInit, BipUp, BipDown, Silent} defetat;
    defetat etat;
    Timer t;
    int i;

    void changeMode(int value);
    void run();
};


Buzzer::Buzzer(){
    TsilenceMs=100;
    etat = BipInit;
    t.reset();
    i=0;
}


/*
 * Initialisation des différents modes de fonctionnement :
 * fonction bip : (400µs+400µs)*50 = 40ms
 *   0 < d <  25  --> 0 : 5 b/s     TsilenceMs 0,16s
 *  25 < d <  50  --> 1 : 4 b/s     TsilenceMs 0,21s
 *  50 < d < 100  --> 2 : 3 b/s     TsilenceMs 0,29s
 * 100 < d < 150  --> 3 : 2 b/s     TsilenceMs 0,46s
 * 150 < d < 200  --> 4 : 1 b/s     TsilenceMs 0,96s
 * 200 < d        --> 5 : 0 b/s     TsilenceMs infini
 */

//Modifie la valeur du temps de silence entre 2 bips en fonction de la distance
void Buzzer::changeMode(int value){
    if(value<=25){                      //0
        TsilenceMs=160;
    }else if(value>25 && value<=50){    //1
        TsilenceMs=210;
    }else if(value>50 && value<=100){   //2
        TsilenceMs=290;
    }else if(value>100 && value<=150){  //3
        TsilenceMs=460;
    }else if(value>150 && value<=200){  //4
        TsilenceMs=960;
    }else{                              //5
        TsilenceMs=1000000000;
    }
}

/*
Si la valeur sort de son intervalle de fonctionnement,
on modifie le mode de fonctionnement courant.
*/

void Buzzer::run(){ // paramètre : t (Timer), i, etat
    switch(etat){

        case BipInit :
```

```
            buzz=0;
            i=0;
            etat = BipUp;
            t.start();
            break;

        case BipUp :
            if(t.read_us()>=400 && i<50){
                buzz=1;
                i++;
                etat = BipDown;
                t.reset();
                t.start();
            }
            if(i>=50){
                etat = Silent;
                t.reset();
                t.start();
            }
            break;

        case BipDown :
            if(t.read_us()>=400 && i<50){
                buzz=0;
                i++;
                etat = BipUp;
                t.reset();
                t.start();
            }
            if(i>=50){
                etat = Silent;
                t.reset();
                t.start();
            }
            break;

        case Silent :
            if(t.read_ms()>=TsilenceMs){
                etat=BipInit;
            }
            break;

        default :
            etat = BipInit;
            break;
    }
}


#endif
```

A.2.4 main.cpp

```
#include "mbed.h"
#include <string.h>
#include "UART_Radar.h"
#include "buzzer.h"

#define TRIG_1  PA_8
#define ECHO_1  PC_7

Serial pc(USBTX, USBRX); //reception --- tx, rx

//Instanciation des GPIOs.
DigitalOut trig(TRIG_1);      //Sortie digitale pour le trigger.
InterruptIn echo(ECHO_1);     //Entrée digitale sur interruption pour l'écho.
InterruptIn position(USER_BUTTON)   //Entrée digitale sur interruption pour le levier de
vitesse.

//Instanciation du Timer.
Timer distance time;    //Timer pour la mesure de distance.
Ticker mesure_time;     //Timer (fixe) avec interruption pour l'acquisition de la distance.
Ticker send_time;       //Timer (fixe) avec interruption pour l'envoi de la distance.
```

```
Ticker bip;            //Timer (fixe) avec interruption pour l'actualisation du buzzer.

//Instanciation des variables.
Buzzer buzzo;
int distance;


/**Fonction qui génère le burst pour la mesure de distance.**/
//But : Générer un burst et reset le timer de distance.
//Donnée :  Le timer distance time est un timer global.
//          trig est un sortie digital.
//                    _______________
// TRIG =>          /               \  ....
//          2 µs          10 µs
void send burst(){
    distance time.reset();
    trig = 0;
    wait_us(2);
    trig = 1;
    wait us(10);
    trig = 0;
}

/**Renvoie le min et le max entre deux entier**/
int max (int a, int b) {  return (a<b)? b : a; }
int min (int a, int b) {  return (a<b)? a : b; }


/********Routine d'interruption sur front montant du signal echo.*********/
//But : Lancer le timer.
//
//Donnée :  Le timer distance_time est un timer global.
//                     ...
// ECHO => ...     /
//                  ^  Start timer.
void echo_rise(){
    distance_time.start();
}

/***********Routine d'interruption sur front descendant du signal echo.******/
//But : Stopper le timer, récupérer la distance en cm
//      et modifier le silence entre deux bip du buzzer
//
//Données:  Le timer distance time est un timer global.
//          L'entier distance est une variable global.
//          Le buzzer buzzo est un objet global.
//                    ____________________________________
// ECHO => _________/      Distant = Temps / 29/2          \__.... Stop timer.
//                                                         ^
//
void echo_fall(){
    distance_time.stop();  //Stoppe le timer.

    distance = distance_time.read_us() /29 / 2; //Convertit le temps (µs) en distance (cm).
    distance = min(distance,230);   //Définit la valeur max de la distance à 230.
    distance = max(distance,0);     //Définit la valeur min de la distance à 0.

    buzzo.changeMode(distance);     //Mise à jour de l'intervalle entre chaque bip
}


/*******Routine d'envoi de la distance via l'uart.***************/
//But : Transformer la distance en chaine de caractères.
//      Envoyer via la fonction send() et attendre un acquittement.
//      Error si on reçoit 3 non-acquittement successif ou atteinte du timeout.
//
//Données:  L'entier distance est une variable global.
//          La chaine message contient la distance sur 3 caractère + '\0'.
//          L'envoie est répété 3 fois quand il y a erreur. Si 3 erreurs, arrêt des émission.
//          Il y a erreur si :  receive ne reçoit pas d'acquittement dans un temps impartie.
//                              contrôle renvoie 0, qui correspond à une erreur de checksum.
void transmit data(){

    int essai, err;
    char message[4];
    char trame[2];
    message[0]=48+(distance/100)%10;
```

```
    message[1]=48+(distance/10)%10;
    message[2]=48+distance%10;
    message[3]=0;

    essai=0;
    send(message);


    pc.printf("%c%c%c\n\r",message[0],message[1],message[2]);
    err=receive(trame);                     //Bug avec le nouvelle affichage
    pc.printf("error%d\n\r",err);

    while((controle(trame)==1 || err!=0) && essai < 3){
        essai++;
        if( essai >= 3) while(1);
        send(message);
        err=receive(trame);
        pc.printf("error%d\n\r",err);
    }
}

/********Routine d'interruption sur front montant du signal pos.*********/
//But : Stopper les routines en cours (Marche avant enclenchée)
void position_rise(){

        echo.rise(NULL);   //Routine NULL sur interruption de echo en front montant.
        echo.fall(NULL);   //Routine NULL sur interruption de echo en front descendant.
        mesure_time.detach();    //Stoppe la routine send_burst.
        send_time.detach();      //Stoppe la routine transmist_data.
        bip.detach();            //Stoppe la routine l'actualisation du buzzer.
}


/********Routine d'interruption sur front descendant du signal pos.*********/
//But : Lancer le programme (Marche arrière enclenchée)
void position_fall(){
        echo.rise(&echo_rise);   //Routine sur interruption de echo en front montant.
        echo.fall(&echo_fall);   //Routine sur interruption de echo en front descendant.
        mesure time.attach(&send burst, 0.5);       //Lance send burst toutes les 500ms.
        wait(0.1);                                  //Décalage entre send et transmit.
        send_time.attach(&transmit_data, 0.5);      //Lance transmist_data toutes les 500ms.
        wait(0.1);                                  //Décalage entre transmit et buzz.
        bip.attach us(&buzzo, &Buzzer::run, 400);   //Lance l'actualisation du buzzer toutes
les 400µs ( F = 2.5 kHz ).
}


int main() {
        position.rise(&position rise); //Routine sur interruption de pos en front montant.
        position.fall(&position fall); //Routine sur interruption de pos en front descendant.
    while(1);
}
```

B. Gestionnaire de l'écran

main.c

```
#include "stm32f4xx.h"                  // Device header
#include "stm32f4xx_rcc.h"              // Keil::Device:StdPeriph Drivers:RCC
#include "stm32f4xx_usart.h"            // Keil::Device:StdPeriph Drivers:SPI
#include "GPIO_STM32F4xx.h"             // Keil::Device:StdPeriph Drivers:GPIO
#include "GLCD.h"                       // Keil::Board Support:GLCD

#include <stdio.h>
#include <string.h>


//Constante :img contient l'image de la voiture.
//Taille : 30x28 pixels
//Couleur :
// 0 = Blanc
// 1 = Noir
// 2 = Rouge
// 3 = Gris
// 4 = Bleu
// 5 = Jaune
const short img[30][28] = {
{ 0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0 },
{ 0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,1,1,0,0,0,0,0,0,0,0 },
{ 0,0,0,0,0,0,0,0,0,1,1,2,0,0,0,0,0,0,1,1,0,0,0,0,0,0 },
{ 0,0,0,0,0,0,0,0,1,2,2,0,0,0,0,0,2,2,2,0,1,0,0,0,0,0 },
{ 0,0,0,0,0,0,0,1,2,2,0,0,0,0,0,2,2,2,2,2,0,1,0,0,0,0 },
{ 0,0,0,0,0,0,0,1,2,0,0,0,0,0,2,2,2,2,2,2,2,1,0,0,0,0 },
{ 0,0,0,0,0,0,1,0,0,0,0,0,0,0,2,2,2,2,2,2,2,0,1,0,0,0 },
{ 0,0,0,0,0,0,1,0,0,0,0,0,0,0,2,2,2,2,2,2,2,0,1,0,0,0 },
{ 0,0,0,0,0,0,1,0,0,0,0,0,0,0,2,2,2,2,2,2,2,0,0,2,1,0 },
{ 0,0,0,0,0,0,0,1,0,1,1,0,0,0,0,2,2,2,0,0,0,2,1,0,0,0 },
{ 0,0,0,0,0,0,0,1,1,5,5,1,1,0,0,0,0,0,0,0,0,1,0,0,0,0 },
{ 0,0,0,0,0,0,0,0,1,5,5,1,5,1,1,0,0,0,0,0,0,0,1,0,0,0 },
{ 0,0,0,0,0,0,0,0,1,5,5,1,5,5,5,1,0,0,0,0,0,0,1,0,0,0 },
{ 0,0,0,0,0,0,0,0,1,5,5,5,5,5,5,1,0,0,0,0,0,1,0,0,0,0 },
{ 0,0,0,0,0,0,0,0,0,1,1,1,5,5,5,5,5,1,0,0,1,0,0,0,0,0 },
{ 0,0,0,0,0,0,0,0,0,1,5,5,5,5,5,5,1,1,1,1,1,0,0,0,0,0 },
{ 0,0,0,0,0,0,0,0,0,0,1,5,1,1,1,5,1,4,1,0,0,0,0,0,0,0 },
{ 0,0,0,0,0,0,0,0,0,1,1,5,5,5,5,5,5,1,4,1,0,0,0,0,0,0 },
{ 0,0,0,0,0,0,0,0,0,1,5,5,5,5,5,5,5,1,4,4,1,0,1,1,0,0 },
{ 0,0,0,0,0,0,0,0,1,5,5,5,5,5,5,5,1,4,4,4,1,1,2,2,1,0 },
{ 0,0,0,1,1,1,1,1,1,1,1,5,5,1,1,4,4,4,4,4,1,2,2,1,0,0 },
{ 0,0,1,3,3,3,1,2,2,2,2,1,1,1,4,4,4,4,4,4,1,2,3,3,1,0 },
{ 0,1,3,1,1,1,1,2,2,2,2,0,0,0,1,4,4,4,4,4,4,1,1,1,1,0 },
{ 0,1,1,3,3,1,1,1,2,2,2,2,0,0,1,4,4,4,4,4,1,1,1,1,1,0 },
{ 0,1,3,1,1,3,1,1,2,2,2,2,0,0,0,1,1,1,1,3,1,1,1,3,3,1 },
{ 1,1,1,3,3,1,1,1,1,1,1,1,1,1,3,3,3,3,0,1,1,3,1,1,3,1 },
{ 1,1,3,1,1,3,1,1,3,3,3,3,3,3,3,3,3,0,0,2,1,1,3,1,0,3 },
{ 1,1,3,1,0,3,1,1,0,0,0,0,0,0,0,0,2,2,2,1,1,3,0,0,3,1 },
{ 0,1,1,3,3,1,1,0,1,1,1,1,1,1,1,1,1,1,1,0,1,1,3,3,1,1 },
{ 0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0 }};


//Prototypes des fonctions
void USART1_Initialize(void);
void GPIO_Initialize(void);
int USART1_transmitter_empty(void);         //Teste l'état du canal d'émission
int USART1_receiver_ready(void);            //Teste l'état du canal de réception
void USART1_puts(char *s);                  //Emet une chaine de caractères.
void format(char *msg, char *msg_formated); //Formate la trame de données à émettre.
void send(char *data);                      //Formate et envoie une trame de données
bool receive(char *frame);                  //Reçoit une chaine de caractères

int trametoint(char trame[3]);              //Convertit un tableau de char en int

void makeVecteur(int distance, char vecteur[60]); //Vecteur graphique symbolisant la distance restante
void draw_voiture(int x0 , int y0);         //Affiche la voiture à l'écran
void clean_voiture(int x0 , int y0);        //Efface la voiture


/**********************Initialisation de L'UART**********************/
void USART1_Initialize(void){
    USART_InitTypeDef usart1_Conf;

    //Enable clock for USART1
    RCC_APB2PeriphClockCmd( RCC_APB2Periph_USART1 , ENABLE );


    // Initialisation UART
    usart1_Conf.USART_BaudRate = 9600;
    usart1_Conf.USART_WordLength = USART_WordLength_8b;
    usart1_Conf.USART_StopBits = USART_StopBits_1;
```

```
    usart1_Conf.USART_Parity = USART_Parity_No;
    usart1_Conf.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
    usart1_Conf.USART_HardwareFlowControl = USART_HardwareFlowControl_None;

    USART_Init(USART1, &usart1_Conf);
    USART_Cmd(USART1, ENABLE);
}

/**********************Initialisation des GPIOs**********************/
void GPIO_Initialize(void){
    //Enable clock for GPIO
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);

    //Configuration du GPIO
    // Broche USART1 :
    // CK = PB3
    // TX = PB6
    // RX = PB7
     GPIO_PinConfigure(GPIOB,6,GPIO_MODE_AF,GPIO_OUTPUT_PUSH_PULL,GPIO_OUTPUT_SPEED_100MHz,GPIO_PULL_UP);
     GPIO_PinConfigure(GPIOB,7,GPIO_MODE_AF,GPIO_OUTPUT_PUSH_PULL,GPIO_OUTPUT_SPEED_100MHz,GPIO_PULL_UP);
     GPIO_PinConfigure(GPIOB,3,GPIO_MODE_AF,GPIO_OUTPUT_PUSH_PULL,GPIO_OUTPUT_SPEED_100MHz,GPIO_PULL_UP);

    //Active le mode Alternate Function
    GPIO_PinAF (GPIOB,6,GPIO_AF_USART1);
    GPIO_PinAF (GPIOB,7,GPIO_AF_USART1);
    GPIO_PinAF (GPIOB,3,GPIO_AF_USART1);
}

/***********Teste si le caractère a bien été émis*****************/
// Renvoie 1 si le flag TXE (TXEnable) de l'uart1 est à 1.
int USART1_transmitter_empty(void){ return (USART_GetFlagStatus(USART1, USART_FLAG_TXE) == SET) ? 1 : 0; }

/************Teste si un caractère a été reçu.**********************/
// Renvoie 1 si le flag RXNE (TXNotEnable) de l'uart1 est à 1.
int USART1_receiver_ready(void){ return (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) == SET) ? 1 : 0; }

/************Emission de la chaine de caractère.*******************/
//Entrée : *text = Pointeur vers la chaine de caractères a emetre.
void USART1_puts(char *text){
    while(*text != '\0'){   //Emission des caractères jusqu'au caractère \0 (Fin de chaine).
        if(USART1_transmitter_empty()){ //Transmission si aucun caractère n'est en attente
            USART_SendData(USART1, *text); //Emission des caractère un à un.
            text++;
        }
    }
}

/*Formate la chaine de caractère pour correspondre à notre protociole de transmission.*/
  Entrées : *msg = Pointeur vers une chaine de caractères contenant notre message à formater.
   *msg_formated = Pointeur vers une chaine de caractères formaté.
   Format =>  STX + MSG + CHK + ETX
*/
void format(char *msg, char *msg_formated){
    char STX = '\x02';  //Caractère Start Of Frame
    char ETX = '\x03';  //Caractère End Of Frame
    char CHK;           //Caratére de checksum

    int somme=5;        //somme = STX + ETX = 5
    int i=0;
    while(msg[i]!='\0'){
        somme= somme + msg[i];
        msg_formated[1+i] = msg[i];  //Tranfert des caractères de msg vers msg_formated
        i++;
    }
    if(somme>255){
        somme++;
    }
    somme=somme%256;
    CHK = ~somme;

    //Concaténation des différents caractères utiles pour notre protocole
    msg_formated[0] = STX;    msg_formated[i+1] = CHK;
    msg_formated[i+2] = ETX;
    msg_formated[i+3] = '\0';   // *msg_formated est une chaine de caractères. Doit donc recevoir un \0 en
fin de chaine.
}


/*********Emission d'une chaine de caractères**************/
void send(char *data){
    char* trame;
    format(data, trame);    //Formatage de la chaine de caractères.
    USART1_puts(trame);     //Emission de la chaine de caractères.
}

/************Fonction de réception de chaine.*************/
//Entrée : *frame = Pointeur vers la variable dans laquel sera stocké la chaine de caractères recue.
//Réception sans timeout => !! Fonction bloquante !!.
```

```
bool receive(char *frame){
    int i=0;
    unsigned int sommeBuffer;
    while(1){
        char r;
        if(USART1_receiver_ready()){        //Attente de la réception d'un caractère.
            r = USART_ReceiveData(USART1);  //Récupération de la valeur du caractère.
            switch(r){
                case 2:     //Si le caractère est STX.
                    sommeBuffer=2;      //Somme = 2
                    break;

                case 3:     //Si le caractère est ETX.
                    sommeBuffer+=3;     //Ajoute 3 à la somme.

                    if(sommeBuffer>255){ //Calcul de checksum.
                        sommeBuffer++;
                    }
                    sommeBuffer=~sommeBuffer;
                    sommeBuffer=sommeBuffer%256;

                    if(sommeBuffer==0){  //Les sommes concordent, les données sont valides.
                        frame[i-1]='\0'; //On remplace le dernier caractère (Checksum) par \0.
                        return true;     //Pas d'erreur
                    }else{               //Message invalidé par la somme.
                        return false;    //Erreur
                    }
                    break;

                default :   //Tout autre caractères
                    frame[i]=r;             //Ajoute à la chaine de caractères
                    sommeBuffer+=(int)r;    //Somme = Somme + caractères
                    i++;
                    break;
            }
        }
    }
}

/*************Création d'un vecteur d'évolution de la distance.*************/
//Entrées :  distance = distance à symboliser
//           vecteur[60] = pointeur vers un vecteur de 60 caractères.
void makeVecteur(int distance, char vecteur[60]){
    int i=0;
    for( i=1 ; i<=59 ; i++){   //Pour chaque caractères…
        //… si la distance est inférieure à position*5, le caractères faut ' '.
        if( i*5 <= distance) vecteur[i-1]=' ';
        else vecteur[i-1]='X';  //… sinon le vecteur comprend un X par tranche de 5cm.
    }
}

/***********Convertit une chaine de caractères en entier.***************/
//Entrée : trame[3] = Pointeur vers tableau de 3 caractères.
//Retour : Valeur de la chaine de caractères en entier.
int trametoint(char trame[3]){
    return 100*(trame[0]-48) + 10*(trame[1]-48) + (trame[2]-48);
}


/*******Dessine la voiture décrite dans la contante img.***********/
//Entrée : x0, y0 = Possition de l'image.
void draw_voiture(int x0 , int y0){
    int l,c;
    for(c = 0 ; c<28 ; c++){
        for(l = 0 ; l<30 ; l++){
            if(img[l][c] == 0){ GLCD_SetTextColor(White); } //Choix des couleurs.
            if(img[l][c] == 1){ GLCD_SetTextColor(Black); }
            if(img[l][c] == 2){ GLCD_SetTextColor(Red); }
            if(img[l][c] == 3){ GLCD_SetTextColor(LightGrey); }
            if(img[l][c] == 4){ GLCD_SetTextColor(Blue); }
            if(img[l][c] == 5){ GLCD_SetTextColor(Yellow); }

            GLCD_PutPixel(x0 + 2*c , y0 + 2*l);     //Desine chaque pixel 4 fois ( Zoom x 2 ).
            GLCD_PutPixel(x0 + 2*c+1 , y0 + 2*l);
            GLCD_PutPixel(x0 + 2*c , y0 + 2*l+1);
            GLCD_PutPixel(x0 + 2*c+1 , y0 + 2*l+1);
        }
    }
}

/***********Efface la voiture.***************/
void clean_voiture(int x0 , int y0){
    int l,c;
    GLCD_SetTextColor(White);   //Couleur de la gomme = Blanc.
    for(c = 0 ; c<28 ; c++){
        for(l = 0 ; l<30 ; l++){
            GLCD_PutPixel(x0 + 2*c , y0 + 2*l);
            GLCD_PutPixel(x0 + 2*c+1 , y0 + 2*l);
```

```
            GLCD_PutPixel(x0 + 2*c , y0 + 2*l+1);
            GLCD_PutPixel(x0 + 2*c+1 , y0 + 2*l+1);
        }
    }
}

int main(){
    char vecteur[60] = "                                                          \0";
    char data[3];
    int dist = 0;
    int i;

    GLCD_Initialize();          //Initialisation de l'écran LCD.
    GLCD_WindowMax();           //Definie la taille maximum de l'écran.
    GLCD_Clear(White);          //Efface l'écran (Affichage Blanc.
    GLCD_SetBackColor(White);   //Definie la couleur de fond à blanc.


    draw_voiture(2,110); //Dessine la voiture à la position x=2 , y=110. (Valeur en pixel).

    GLCD_SetTextColor(Red);                         //Change la couleur d'écriture en rouge.
    GLCD_DisplayString(1,3,1,"Radar de recul");
    GLCD_SetTextColor(Black);                       //Change la couleur d'écriture en noir.
    GLCD_DisplayString(3,1,1,"Distance :     cm");
    USART1_Initialize();    //Initialisation de l'UART
    GPIO_Initialize();      //Initialisation des GPIOs

    while(1){
        if(receive(data)){              //Attente de la réception d'un message.
            send("\x06");               //Si la réception est valide, envoie un acquittement.
            dist = trametoint(data);    //Convertie la trame en entier.
            GLCD_SetTextColor(Black);   //Change la couleur d'écriture en noire.
            GLCD_DisplayString(3,12,1,data);    //Ecrit la distance ( 3 caractères ) à la position x = 3 ,
y = 12 (Valeur en caratére) et avec la police 1 (Grande).
            data[0] = 0;
            makeVecteur(dist,vecteur);  //Création du vecteur symbolisant la distance.
            GLCD_DisplayString(17,11,0,vecteur); //Ecrit le vecteur de distance sur 4 ligne.
            GLCD_DisplayString(18,11,0,vecteur); //Taille des caractères 0
            GLCD_DisplayString(19,11,0,vecteur);
            GLCD_DisplayString(20,11,0,vecteur);

            if(dist <= 17){         //Si la distance <= 17 cm, affiche "Attention choc !"
                GLCD_SetTextColor(Red);
                GLCD_DisplayString(8,2,1,"Attention choc !");
                GLCD_SetTextColor(Black);
            }
            else if( dist >= 200 ){ //Si la distance >= 200 cm, affiche "Out of range"
                GLCD_SetTextColor(Red);
                GLCD_DisplayString(8,2,1,"  Out of range ");
                GLCD_SetTextColor(Black);
            }else{    //Sinon affiche rien.
                GLCD_DisplayString(8,2,1,"                ");
            }

        }
        else{
            send("\x21");   //Si la réception n'est pas valide renvoie un non acquittement.
            GLCD_SetTextColor(Red);
            GLCD_DisplayString(8,2,1,"  Data Error ! ");    //Affiche aussi Data Error !
            GLCD_SetTextColor(Black);
        }
    }
}
```