



**MASTER 1 ELECTRONIQUE – ENERGIE ELECTRIQUE -
AUTOMATIQUE**

PARCOURS : ELECTRONIQUE DES SYSTÈMES EMBARQUÉS ET
TÉLÉCOMMUNICATION (ESET)

Rapport de synthèse du projet :

Simulateur de communication entre OBC et la
charge utile nano-satellite NIMPH

Réalisé par :

Boutlalek Hicham

Djibril Abdel-Waris

Soulaimani Marouane

Responsable du suivi :

Mr. Frédéric Camps LAAS/CNRS

Table des matières

Liste des figures

I.	Introduction	1
II.	Contexte générale du projet et problématique.....	2
III.	Initiation avec sysML.....	3
1.	Définition	3
2.	Diagramme SysML de séquence (Sequence Diagram).....	3
3.	Diagramme SysML d'états « stm » (State Machine Diagram)	5
3.1.	Notions de base	5
3.2.	Etat – transition	5
IV.	Architecture du système	6
V.	Présentation de la carte OLIMEX	7
1.	Carte Olimex	7
2.	Accessoire de la carte	8
2.1.	Programmateurs JTAG	8
2.2.	Câble série RS232.....	9
VI.	Protocole de communication I2C et SPI	10
1.	Bus de communication I2C	10
1.1.	Transmission d'adresse sur I2c	11
1.2.	Ecriture de donnée sur I2c	11
1.3.	Gestion de Conflits sur I2C.....	12
2.	Protocole de communication SPI.....	12
VII.	Développement des programmes	13
1.	Modélisation sysML du système.....	13
2.	Codes utilisés pour l'échange entre les cartes	14
3.	Présentation du côté Arduino (OBC)	16

4.	Présentation du côté Olimex (Atmega128A)	17
VIII.	Conclusion	21
Annexes		

Liste des figures

Figure 1: schéma représente le déroulement de la communication Réelle	2
Figure 2: Structure d'un diagramme de séquence.....	4
Figure 3: Structure de diagramme d'état	5
Figure 4: OBC utilise l'interface d'EDMON pour le piloter	6
Figure 5: Schéma bloc du système	7
Figure 6: La carte OLIMEX.....	8
Figure 7: Programmeur AVR – JTAG	8
Figure 8: Adaptateur JTAG - AVR et JTAG-SAM	9
Figure 9: Câble série RS232.....	9
Figure 10: Interface PuTTY	10
Figure 11: Chronogramme de communication I2C - REPEATED START and STOP conditions	11
Figure 12: Chronogramme de communication I2C - circulation d'adresse sur le bus	11
Figure 13: Chronogramme de communication I2C - Circulation des données sur le bus.....	11
Figure 14: Chronogramme de communication I2C - Transmission Adresse suivi par data	12
Figure 15: Communication à travers le bus SPI entre un maître et plusieurs esclaves.....	13
Figure 16: Messages indiquant l'écriture dans les fichiers texte	16
Figure 17: Définition des codes de Status de l'I2C dans le fichier i2c_status.h.....	17

I. Introduction

Dans le cadre de l'UE Initiation à la recherche et projet, nous avons été amenés à travailler sur le projet intitulé : Simulateur de communication entre OBC (On Board Computer) et la charge utile nanosatellite **NIMPH**. Ce projet s'inscrit dans le projet du nanosatellite **NIMPH**, dans le cadre du projet JANUS piloté par le CNES.

NIMPH est l'acronyme de Nanosatellite to Investigate Microwave Photonics Hardware. C'est un projet dédié à démontrer la fiabilité d'un système Opto-Micro-onde dans l'espace.

Le but de ce projet d'étude est d'investir les étudiants dans la conception de la charge utile, afin d'assurer la communication entre l'OBC et la charge utile EDMON (Erbium Doped fiber Monitoring), la charge utile principale qui comprend les composants optiques et l'électronique de contrôle et de mesures associées.

Dans ce rapport, nous allons présenter le travail que nous avons fait durant l'année d'étude, en commençant par la présentation du contexte général et la problématique du projet. Pour la description du déroulement de la communication entre OBC (représenté par ARDUINO MEGA) et la charge utile EDMON (représenté par la carte Olimex-Atmega128A) nous utilisons deux diagrammes SysML qui représentent le diagramme d'état et le diagramme de séquences que nous décrivons dans la partie initiation avec SysML. Ensuite, nous présentons la carte Olimex et ces accessoires. On détaillera le bus I2C et le bus SPI utilisés par l'ARDUINO et on finit avec une démonstration de la communication entre les deux cartes en fonction des différents scénarios.

II. Contexte générale du projet et problématique

Notre objectif était d'assurer la communication entre une carte Arduino mega qui remplace l'OBC (Ordinateur de bord) qui est en cours de conception, utilisée pour recevoir des scenarios de communication avec une autre carte Olimex qui joue le rôle de la charge utile. Le choix de cette carte vient du choix du microcontrôleur Atmega128A intégré dans la carte du nanosatellite. La communication se fait à travers le bus I2C, donc notre mission est de créer un programme efficace qui assure la communication entre les deux cartes à travers ce bus. Ainsi de créer plusieurs scénarios de tests permettent de vérifier la conformité du système et sa stabilité.

Les scénarios seront envoyés depuis la terre vers le nanosatellite où l'OBC les utilise. Ils sont écrits en fichier texte et contiennent des lignes de commande qui seront envoyés par l'OBC à la charge utile sur le bus I2C. La charge utile s'occupe d'une tâche selon la commande reçue, soit de répondre directement ou s'occupe de la réalisation de mesure. La figure suivante donne une vision réelle de cette opération.

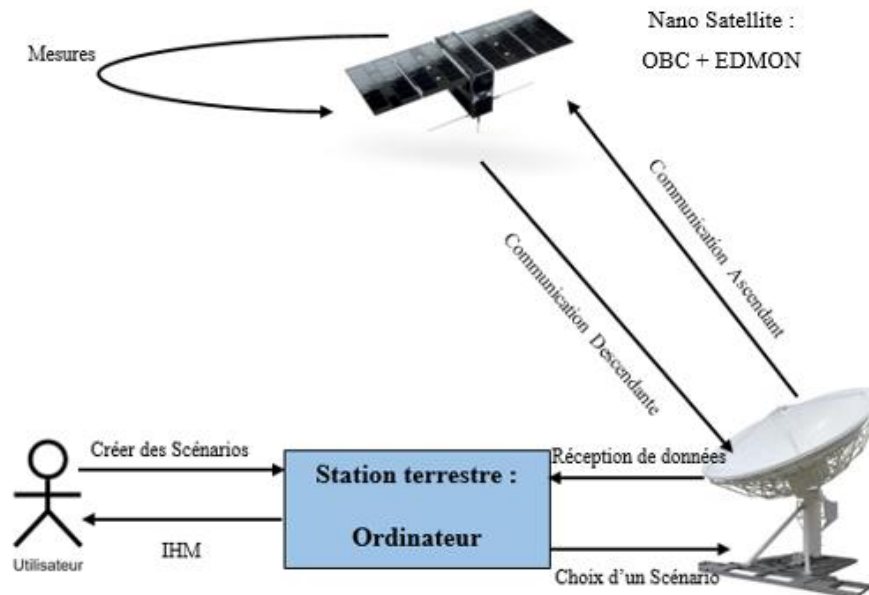


Figure 1: schéma représente le déroulement de la communication Réelle

Comme le nanosatellite est en cours de conception, la réalisation de teste comme sur la figure ci-dessus n'est pas possible. Nous remplaçons la communication terre-Satellite par un module de carte mémoire où on vient de stocké les scénarios sous forme de fichier texte, l'Arduino nous affiche les fichiers existants dans la carte mémoire et nous demande de choisir un scénario de

test pour qu'il soit exécuté. La lecture du fichier texte se fait à travers le bus SPI et c'est entre l'Arduino mega et l'adaptateur de carte mémoire MicroSD.

Pour la description de la communication entre OBC et EDMON nous utilisons des diagrammes sysML de séquences et d'états que nous présentons dans la partie qui suit.

III. Initiation avec sysML

1. Définition

Le SysML (Systems Modeling Language) est un langage de modélisation graphique utilisé dans le domaine de l'ingénierie système qui permet la description, l'analyse, la conception et la validation de système.

SysML s'articule autour de neuf diagrammes dédiés à la représentation de concepts particuliers d'un système.

Dans la suite nous allons détailler 2 diagrammes qui sont des diagrammes comportementaux :

- Diagramme de séquence
- Diagramme d'état

2. Diagramme SysML de séquence (Sequence Diagram)

- Notion de base

Le diagramme de séquence est un diagramme comportemental qui montre chronologiquement la séquence verticale des messages passés entre éléments (lignes de vie) au sein d'une interaction.

Ce diagramme décrit les scénarios correspondant aux cas d'utilisation. Un cas d'utilisation étant décrit par au moins un diagramme de séquence.

Un diagramme de séquence est constitué de :

- Ligne de vie :

Représentation de l'existence d'un élément participant dans un diagramme de séquence. Une ligne de vie possède un nom et un type. Elle est représentée graphiquement par une ligne verticale en pointillés.

- Activation d'une ligne de vie :

Les bandes verticales (bandes en bleu sur l'exemple) le long d'une ligne de vie représentent des périodes d'activation. Elles sont optionnelles, mais permettent de mieux comprendre la flèche pointillée du message de retour.

- Messages :

Ce sont des éléments de communication unidirectionnelle entre les lignes de vie qui déclenchent une activité dans le destinataire. La réception d'un message provoque un événement chez le récepteur. Il existe deux types de messages, les messages synchrones et les messages asynchrones.

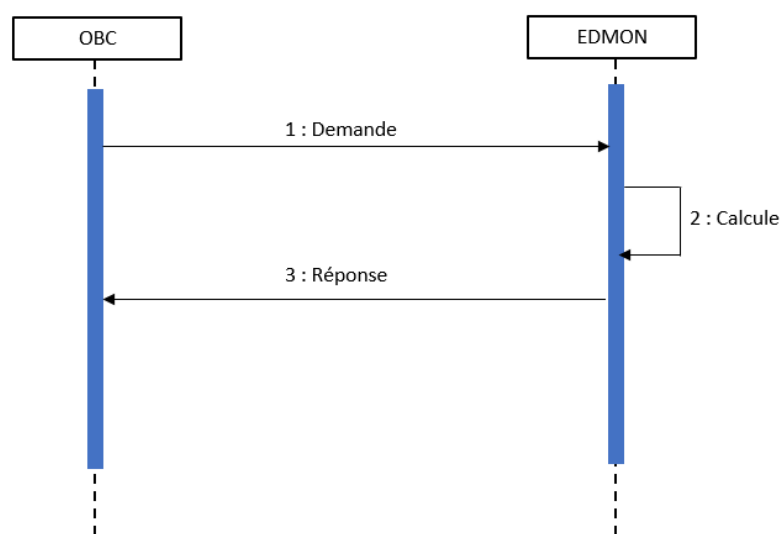


Figure 2: Structure d'un diagramme de séquence

SysML propose une notation très utile : le fragment combiné. Chaque fragment possède un opérateur et peut être divisé en opérandes. Les principaux opérateurs sont :

- **loop** : boucle. Le fragment peut s'exécuter plusieurs fois, et la condition de garde explicite l'itération.
- **opt** : optionnel. Le fragment ne s'exécute que si la condition fournie est vraie.
- **alt** : fragments alternatifs. Seul le fragment possédant la condition vraie s'exécutera.
- **par** : fragments parallèles. Permet d'exécuter plusieurs opérandes en parallèle.
- **ref** : permet de créer un hyperlien graphique avec un autre diagramme de séquence.

SysML permet d'ajouter des contraintes temporelles sur le diagramme de séquence.

- **La contrainte de durée** : permet d'indiquer une contrainte sur la durée exacte, la durée minimale ou la durée maximale entre deux événements.

- **La contrainte de temps** : permet de positionner des labels associés à des instants dans le scénario au niveau de certains messages et de les relier ainsi entre eux.

3. Diagramme SysML d'états « stm » (State Machine Diagram)

3.1. Notions de base

Le diagramme d'état décrit les transitions entre les états et les actions que le système ou ses parties réalisent en réponse à un événement.

Il s'agit d'une représentation séquentielle des états d'un système.

Le diagramme d'état se compose :

- Des états
- Des transitions
- Des événements
- Des conditions
- Des effets
- Des activités

3.2. Etat – transition

Un état (state) représente une situation d'un bloc fonctionnel pendant laquelle

- Il satisfait une certaine condition
- Il exécute une certaine activité

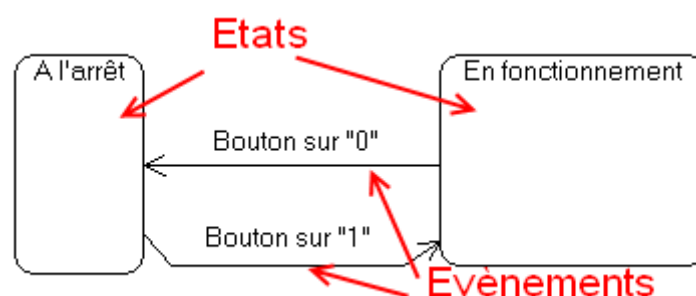


Figure 3: Structure de diagramme d'état

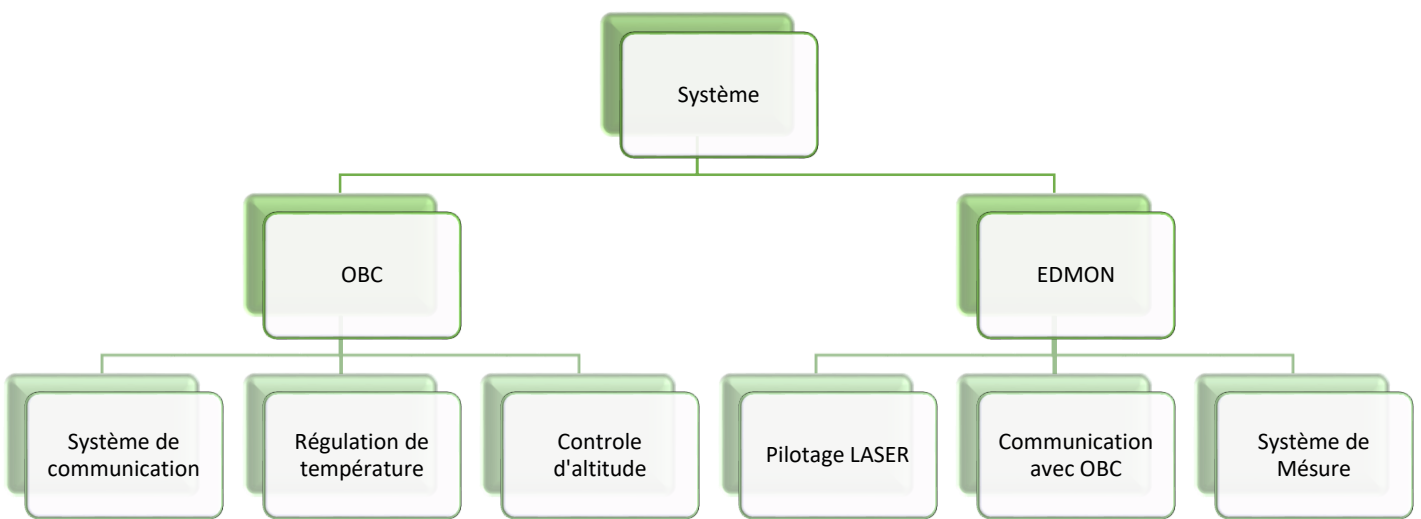
Les différents états sont reliés entre eux par des transitions.

Une transition est liée à un événement. Lorsque l'événement se produit, il peut provoquer le changement d'état de la fonction qui lui est liée. Une transition peut être complétée par une condition notée entre croche, des paramètres notés entre parenthèses et aussi d'une activité, la syntaxe complète de la transition serait : **événement [condition] (paramètres)/activité**.

IV. Architecture du système

Le système que nous sommes en train d'étudier est constitué de deux parties essentielles :

- OBC : C'est l'ordinateur de bord qui est responsable du pilotage de la charge utile et il possède tous les systèmes de communications ainsi les systèmes de régulation de température et de l'attitude.
- EDMON: c'est la charge utile principale qui comprend les composants optiques et l'électronique de contrôle et de mesures associées.



Nous nous occupons de la communication entre l'OBC et l'EDMON. La communication entre eux se fait à travers l'interface de communication offerte par EDMON. Ce dernier utilise l'interface I2C (Two Wire Interface), où l'OBC qui gère le bus (Master) et l'EDMON (Slave) répond à la demande du Master. Le protocole de communication peut être représenté par la figure suivante.

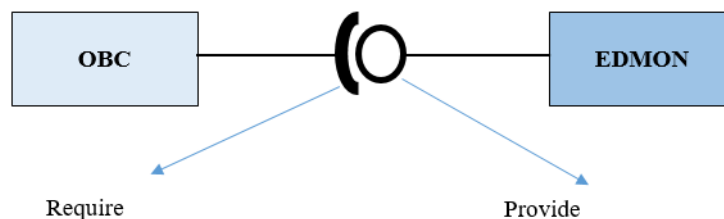


Figure 4: OBC utilise l'interface d'EDMON pour le piloter

- Require : ça veut dire OBC utilise une interface de communication
- Provide : EDMON fournit l'interface de communication

Comme les scénarios de tests seront stocké dans une carte mémoire, et la communication entre l'OBC et la carte mémoire se fait à travers l'interface SPI offerte par l'adaptateur MicroSD. Le schéma bloc de la communication devient comme suite.

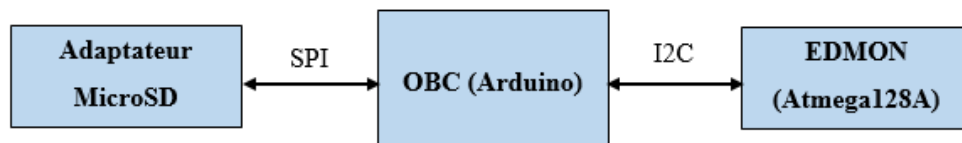


Figure 5: Schéma bloc du système

Dans la suite nous allons présenter la carte Olimex et les éléments nécessaires pour qu'elle puisse communiquer.

V. Présentation de la carte OLIMEX

1. Carte Olimex

Dans notre étude nous avons utilisé la carte OLIMEX (L'AVR-MT128), qu'est une carte simple mais performante dotée du microcontrôleur ATmega128A de Atmel. Elle contient des boutons de commutation, un relais et une grande variété d'interfaces. Elle nécessite une alimentation continue de 12V pour qu'elle fonctionne. Le choix de cette carte vient du choix du microcontrôleur Atmega128A intégré dans la carte du nanosatellite.

L'ATmega128/128A est un microcontrôleur CMOS 8 bits à faible consommation basé sur l'architecture RISC (*Reduced Instruction Set Computer*) améliorée AVR. En exécutant des instructions puissantes en un seul cycle d'horloge, l'ATmega128/128A atteint des débits proches de 1 MIPS par MHz, ce qui permet au concepteur du système d'optimiser la consommation d'énergie par rapport à la vitesse de traitement. Le cœur de l'AVR combine un riche jeu d'instructions avec 32 registres de travail à usage général. Tous les 32 registres sont directement connectés à l'unité logique arithmétique (ALU), ce qui permet d'accéder à deux registres indépendants en une seule instruction exécutée en un cycle d'horloge. L'architecture résultante est plus efficace pour le code tout en atteignant des débits jusqu'à dix fois plus rapides que les microcontrôleurs CISC (*Complex Instruction Set Computer*) conventionnels.

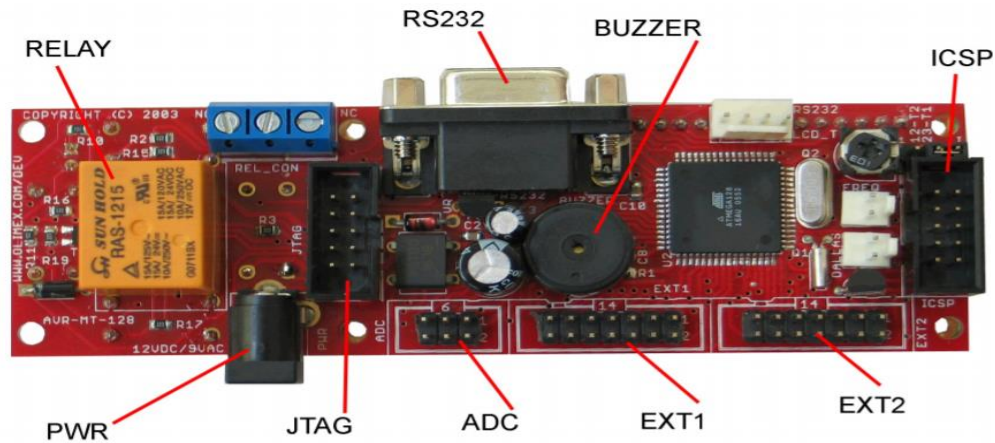


Figure 6: La carte OLIMEX

Le bus I2C que nous utilisons est sur l'extension 1 marqué EXT1 dans la figure ci-dessus. Les pins utilisés sont SCL (horloge de synchronisation) et SDA (Signal de donnée) qui sont liés respectivement aux pins 9 et 10 de l'EXT1, sans oublier de relier les masses de la carte Olimex et d'Arduino.

2. Accessoire de la carte

2.1. Programmeur JTAG

Pour programmer le microcontrôleur ATmega128A de la carte Olimex, on utilise le programmeur JTAG de Atmel puisque la carte ne possède pas de programmeur intégré.



Figure 7: Programmeur AVR – JTAG

Il faut relier la nappe d'une coté avec la sortie AVR, et de l'autre côté avec l'adaptateur suivant :



Figure 8: Adaptateur JTAG - AVR et JTAG-SAM

La sortie JTAG-AVR de l'adaptateur sera reliée avec le port JTAG de la carte Olimex. Nous utilisons le logiciel AtmelStudio pour le développement du programme du microcontrôleur ATmega128A.

2.2. Câble série RS232

Nous utilisons le câble série RS232 pour relier la carte Olimex avec l'ordinateur afin d'afficher des messages, qui permet de donner une vision sur le déroulement de la communication entre les deux cartes du côté de l'ATmega128A, car le logiciel de développement n'a pas une interface homme machine comme. Pour l'Arduino nous pouvons visualiser des messages sur le moniteur série de l'IDE.



Figure 9: Câble série RS232

Du côté d'ordinateur il faut un logiciel qui permet de lire les données présentes dans le port série du PC. Pour cela nous utilisons un logiciel qui s'appelle PuTTY. PuTTY est un émulateur de terminal doublé d'un client pour les protocoles SSH, Telnet, rlogin, et TCP brut. Il permet également des connexions directes par liaison série RS-232. La figure ci-dessous présente l'interface dans laquelle on peut choisir 'Serial' pour qu'il nous ouvre une fenêtre où il affiche les caractères présents sur le port série.

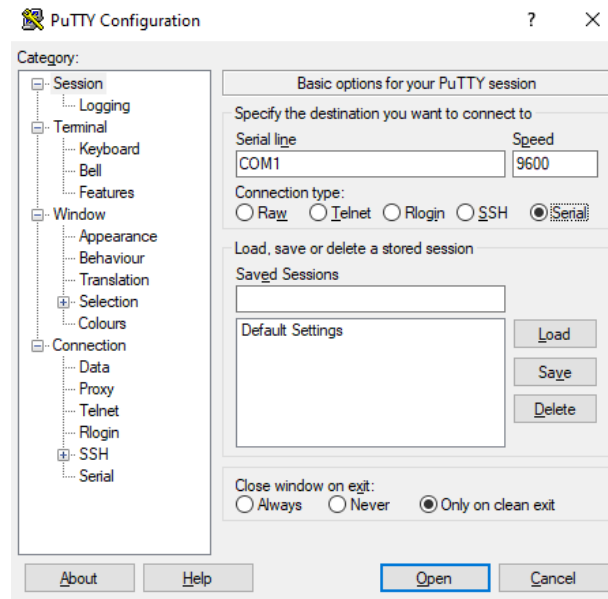


Figure 10: Interface PuTTY

VI. Protocole de communication I2C et SPI

1. Bus de communication I2C

Puisque nous programmons le microcontrôleur ATmega128A pour qu'il puisse communiquer à travers le bus I2C, il est donc indispensable de présenter le bus I2C.

Le bus I2C permet de faire communiquer des composants à travers un protocole de communication série, synchrone, bidirectionnelle et half duplex grâce à seulement 3 fils :

SDA : un signal de donnée ;

SCL : un signal d'horloge ;

Masse : un signal de référence électrique.

Sur le câblage de notre maquette, les deux fils noirs représentent ici le SDA et le SCL ; et le fil blanc le Master reliant la Masse de la carte OLIMEX et celle de la carte Arduino entre elles.

Afin d'effectuer une communication intrinsèque de nos deux cartes, nous avons procédé à la prise de contrôle du bus. Pour cela, il faut qu'au repos le signal de données (SDA) et le signal d'horloge (SCL) soient à 1 (état haut). La transmission de données sur le bus se fait par un passage d'état du SDA à 0 (état bas) ; le SCL reste à 1. L'arrêt de transmission se fait par un retour d'état du SDA et du SCL au repos (respectivement à 1). Lorsqu'un circuit prend le contrôle du bus, il devient le maître. C'est lui qui génère le signal d'horloge.

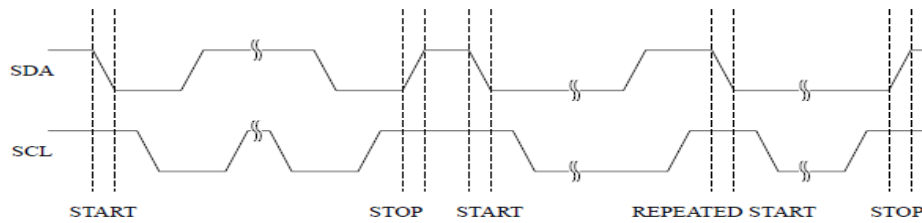


Figure 11: Chronogramme de communication I2C - REPEATED START and STOP conditions

1.1. Transmission d'adresse sur I2c

Ici il est fréquent de connecter plusieurs composants entre eux, d'où l'importance de définir de définir pour chaque composant une adresse unique. L'adresse d'un composant est codée sur 7 bits d'adresses (suivi du bit R/W et ACK) et est définie par son type et par l'état appliqué a un certain nombre de ces broches. Elle est transmise sous la forme d'un octet au format particulier : le bit R/W permet au maitre de signaler s'il veut lire ou écrire une donnée ; le bit d'acquittement ACK fonctionne comme pour une donnée et permet au maitre de vérifier si l'esclave est disponible (état haut).

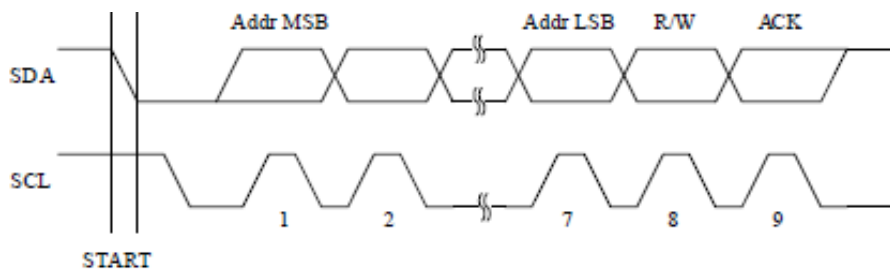


Figure 12: Chronogramme de communication I2C - circulation d'adresse sur le bus

1.2. Ecriture de donnée sur I2c

L'écriture d'un Octet de Data dans certains composants (Mémoire, Microcontrôleur,...) peut parfois prendre du temps et obliger le maitre à attendre l'acquittement ACK avant de passer à l'instruction suivante. Ci-dessous une petite illustration graphique.

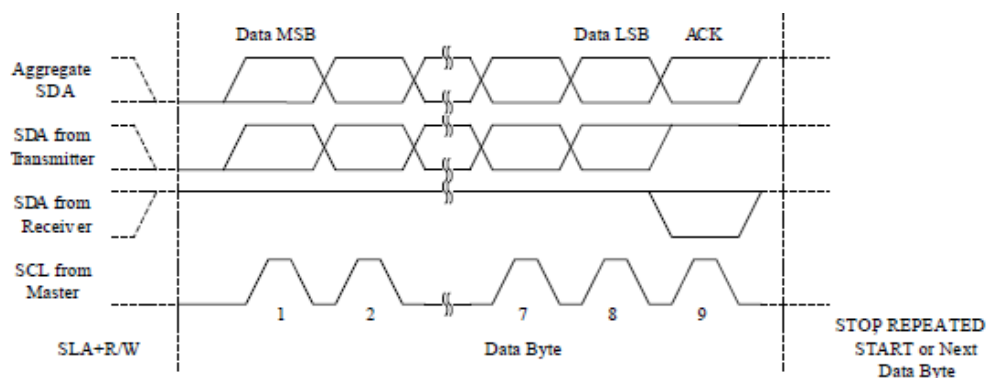


Figure 13: Chronogramme de communication I2C - Circulation des données sur le bus

Ci-dessous le chronogramme d'une transmission typique d'adresse suivis de Data :

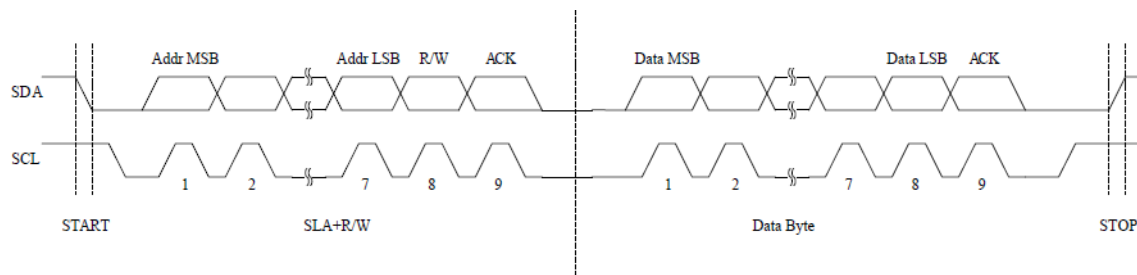


Figure 14: Chronogramme de communication I2C - Transmission Adresse suivi par data

1.3. Gestion de Conflits sur I2C

Pour prendre le control du Bus, un maitre doit vérifier que celui-ci est libre d'abords. Mais néanmoins il peut exister plusieurs maitres sur un même bus simultanément. Ce qui peut créer un conflit durant la communication des cartes :

Chaque circuit vérifie en permanence l'état des lignes SDA et SCL, même s'ils sont eux même en train d'envoyer des données. On distingue alors plusieurs cas :

- Les différents maitres transmettent les mêmes data au même moment : Cas très rare, la transmission est effectuée normalement comme si l'information provenait d'un seul maitre.
- Un des maitres impose un '0' sur le Bus : Cet état '0' passera, puis la transmission continuera. Aucun conflit sur le bus ne sera détecté.
- Un des maitres voulant appliquer un '1' sur le Bus : Si cet état ('1') n'y est plus, c'est qu'un autre maitre aura pris le contrôle du Bus au détriment de celui-ci, qui se rangera pour ne pas perturber la transmission de son collègue. Il recevra néanmoins les données qui lui seront destinées ou envoyées.

2. Protocol de communication SPI

Le protocole SPI est une liaison de communication utilisé pour la transmission synchrone de données entre un maître et un ou plusieurs esclaves. La transmission est en full duplex. Il est utilisé pour la communication rapide de données entre périphériques d'un appareil comme par exemple les mémoires, les systèmes d'affichage, carte SD, etc.

Le bus SPI est composé de deux lignes de données et deux lignes de signal, unidirectionnelles:

- **MOSI** (Master Out Slave In) : Sur la ligne MOSI le maître transmet des données à l'esclave.

- **MISO** (Master In Slave Out) : Sur la ligne MISO l'esclave transmet des données au maître.
- **SCK** (SPI Serial Clock) : Signal d'horloge, généré par le maître, qui synchronise la transmission. La fréquence de ce signal est fixée par le maître et est programmable.
- **SS** (Slave Select) : Ce signal placé au niveau logique 0 permet de sélectionner (adresser) individuellement un esclave. Il y a autant de lignes SS que d'esclaves sur le bus. Le nombre possible de raccordements SS du maître limitera donc le nombre d'esclave.

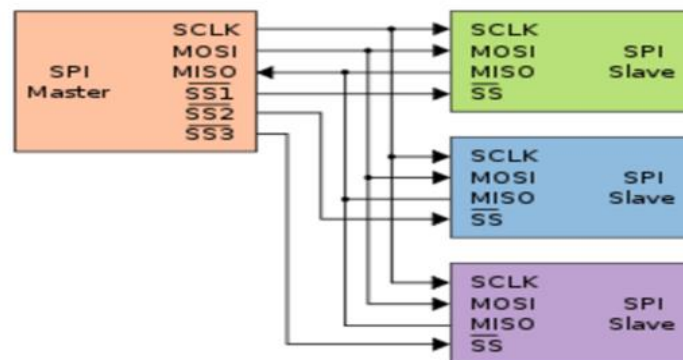


Figure 15: Communication à travers le bus SPI entre un maître et plusieurs esclaves

- Transfert de données :

Après la sélection de l'esclave en mettant sa ligne SS respective au niveau logique 0, à chaque front d'horloge, le maître envoie un bit sur la ligne MOSI. De même, il peut recevoir à chaque front d'horloge, sur la ligne MISO, un bit transmis par l'esclave. La transmission commence par le MSB. La transmission des données est terminée dès que la ligne SS est remise au niveau logique 1.

Le bus SPI est utilisé pour la lecture des fichiers enregistrés dans la carte mémoire. Pour cela nous utilisons la bibliothèque du module de la carte SD "**SD.h**".

VII. Développement des programmes

1. Modélisation sysML du système

Avant d'attaquer la programmation, nous avons réalisé une modélisation de notre système sous forme de diagrammes d'états et de diagramme de séquence comme montré dans les annexes, le logiciel que nous avons utilisé est Modelio v4.1.

Tout d'abord nous avons créé un diagramme de séquence permettant de décrire les échanges entre les différents acteurs (Utilisateur, Carte Olimex, Arduino, Module SD), c'est le

diagramme principal où nous représentons les échanges qui se fait dès que le système est mis en marche jusqu'à le choix d'un scénario par l'utilisateur. Vu le nombre de scénario, on trouve plusieurs choix qu'on ne peut pas les représenté dans un seul diagramme de séquence, la raison pour laquelle nous avons créé des diagrammes de séquence secondaire permettant de décrire les différents scénarios de test que nous avons réalisé.

Comme le diagramme de séquence permet de donner une vision sur les échanges, les diagrammes d'état permet de donner une idée des états à décrire lors de développement de programme. Comme nous avons deux microcontrôleurs à programmer, donc deux diagrammes d'états qu'il faut prévoir pour décrire le comportement de chaque microcontrôleur.

Nous avons traduits les diagrammes d'état en des lignes de programmes ce qui rend le développement plus facile. Les différents diagrammes et programmes que nous avons réalisés sont présents en annexe.

2. Codes utilisés pour l'échange entre les cartes

Sur les fichiers textes, nous enregistrons des lignes de commandes où l'arduino prend une ligne et le traduit en un caractère à envoyer au lieu d'envoyer une chaîne de caractère, cela rend la communication rapide et fluide. Les lignes enregistrées dans les fichiers sont les suivants :

- HELLO
- STARTMEASURE
- GETSTATE
- GETMEASURE
- END

Pour bien gérer la communication entre les deux cartes nous avons utilisé les codes suivants qui seront transmis de la part de la carte Arduino :

- H = **Hello** : cette lettre est le premier qui est envoyé, permet de voir si le Slave est prêt pour commencer une communication.
- M = **Start Measure** : Permet de dire au Slave de commencer les mesures, la carte Olimex ne répond pas à cette code et elle commence les mesures.
- S = **Get State** : L'arduino demande l'état des mesures. Si c'est prêt la réponse sera '**R**' comme Ready, sinon la réponse sera '**N**' comme Not Ready.

- **G = Get Measure** : si la réponse été '**R**' lors de la demande de l'état des mesures. Alors l'envoi de 'G' permet le transfert des données.

La réponse du côté de la carte Olimex dépend du code reçu, et c'est en fonction de l'état elle va répondre par les codes suivants :

- **E** : Pour indiquer une erreur (code male reçu ou inexistant).
- **H** : pour **Hello**, pour dire que l'Olimex est prête pour la communication.
- **N** : pour **Not Ready**, réponse si l'Arduino demande l'état des mesures pour dire que l'Olimex n'a pas encore finis les mesures.
- **R** : pour **Ready**, réponse si l'Arduino demande l'état des mesures pour dire que l'Olimex a finis les mesures et qu'ils sont prêt.

L'Olimex envoi aussi un flux de données représentant les mesures si l'arduino les demandes.

Nous avons créé cinq scénarios, quelques-uns ne sont pas valide et cela pour tester le bons fonctionnement du système. Pour que le système communique il faut impérativement que le scénario commence par HELLO et termine par END, sinon un message d'erreur s'affichera comme quoi le scénario n'est pas valide. Les scénarios sont les suivants :

Scénario1	Scénario2	Scénario3	Scénario4	Scénario5
HELLO END	HELLO GETMEASURE END	HELLO STARTMEASURE GETSTATE END	HELLO STARTMEASURE GETSTATE GETMEASURE END	HELLO GETSTATE GETMEASURE END

Les scénarios 1, 3 et 4 qui sont valident, or les scénarios 2 et 5 ne sont pas valides car il ne faut pas demander les mesures si on ne demande pas de faire les mesures. Or si on demande de faire les mesures et qu'on demande les mesures sans avoir demandé l'état, celui-là n'est pas possible aussi. Dans le scénario 5 on demande l'état des mesures sans avoir demandé de faire les mesures, donc c'est un scénario qui n'est pas valide.

Autre possibilité qu'est valide, c'est de choisir le scénario 3 où on demande de faire les mesures et la demande de l'état de mesures, suivi du scénario 2 ensuite, maintenant il sera possible de recevoir les données car ils sont déjà présents par application du scénario 3.

3. Présentation du côté Arduino (OBC)

L'utilisation du protocole de communication I2C sur l'arduino nécessite l'appel à la bibliothèque Wire.h. Ainsi il faut définir l'adresse du Slave (nous avons choisi 0x09) pour bien communiquer. Nous utilisons aussi deux autres bibliothèques qui sont SPI.h qui nous permet d'utiliser le bus SPI, ainsi la bibliothèque SD.h qui permet à l'arduino de communiquer avec la carte mémoire sur le bus SPI.

Pour enregistrer des fichiers textes dans la carte mémoire nous utilisons une fonction que nous avons développée qui s'appelle 'writeToFile', cette fonction prend en paramètre le nom de fichier et le texte à écrire. L'écriture se fait une seule fois comme présenté dans la figure suivante.

```
COM3
|
Initializing SD card...SD card is ready to use.
Writing to test1.txt is Done.
Writing to test2.txt is Done.
Writing to test3.txt is Done.
Writing to test4.txt is Done.
Writing to test5.txt is Done.
I2C Master Demonstration
Choice the scenareo that you want to execute :
1 : test1.txt
2 : test2.txt
3 : test3.txt
4 : test4.txt
5 : test5.txt
```

Figure 16: Messages indiquant l'écriture dans les fichiers texte

Dans la loop, en se basant sur le diagramme d'état que nous avons réalisé, nous avons créé notre programme pour qu'une liste s'affiche pour le choix d'un scénario parmi les cinq qu'on a. Dès qu'on choisit un fichier, ça vient l'étape de la lecture du fichier, pour cela nous avons créé une fonction qui s'appelle readText qui prend en paramètre le nom de fichier, un pointeur vers un tableau dynamique où le scénario sera stocké, ainsi la taille du tableau.

Ensuite, l'état de vérification où nous regardons la première et la dernière ligne du scénario qu'on vient de choisir, s'il commence par Hello et termine par END, on considère qu'il est valide, sinon un message qui s'affiche pour indiquer que le scénario n'est pas valide et la liste de choix s'affiche de nouveau.

L'état suivant qui est nécessaire, c'est d'envoyer le code H pour voir si la carte Olimex est prête pour commencer un échange. Si oui, on commence l'échange en fonction des lignes de scénario.

A la fin de chaque échange, la liste de choix s’affiche de nouveau permettant de choisir un nouveau scénario. Le programme que nous avons développé est en annexe.

4. Présentation du côté Olimex (Atmega128A)

Le code que nous avons développé en C pour la programmation du microcontrôleur est constitué de deux fichiers. Un fichier main.c qui contient le programme principal qui s’exécute dès la mise sous tension de la carte. Et d’un deuxième que nous appelons i2c_status.h, c’est un fichier header que nous l’appelons sur le main.c, donc joue le rôle d’une bibliothèque et qui contient des déclarations ainsi les messages que nous voyons sur l’ordinateur. Ce fichier est important puisque l’utilisation de bus I2C sur le microcontrôleur nous exige d’utiliser certains codes pour savoir le Statut ou l’état avec laquelle déroule le fonctionnement pendant la communication. Puisque la carte Olimex en mode slave peut recevoir des données ou bien transmettre des données, cela nous donne des codes différents selon la demande du Master. Ci-dessous une capture pour la définition de ces codes, on les donne des noms pour faciliter l’écriture du programme.

```
// SLAVE RECEIVER (SR) CODES
#define SR_SLA_ACK          0x60          // slave address receive and ack send SLA+W
#define SR_ARB_LOST_SLA_ACK 0x68          // Arbitration lost in SLA+R/W and ack send
#define SR_GEN_CALL_ACK     0x70          // General call address received and ack send
#define SR_DATA_ACK         0x80          // data received With SLA_ACK addressable and ack send
#define SR_DATA_NACK        0x88          // data received and No ack send
#define SR_GEN_CALL_DATA_ACK 0x90          // general call data receive and ack send
#define SR_GEN_CALL_DATA_NACK 0x98        // general call data receive and NO ack send

// SLAVE TRANSMITTER (ST) CODES
#define ST_SLA_ACK          0xA8          // slave address receive and ack send SLA+R
#define ST_ARB_LOST_SLA_ACK 0xB0          // Arbitration lost in SLA+R/W and ack send
#define ST_DATA_ACK         0xB8          // data TRANSMITTED and ack RECEIVED
#define ST_DATA_NACK        0xC0          // data TRANSMITTED and NOT ack RECEIVED
#define ST_DATA_ACK_TWEA0   0xC8          // Data byte in TWDR has been transmitted (TWEA = "0"); ACK has been received
#define TW_STOP             0xA0
```

Figure 17: Définition des codes de Status de l'I2C dans le fichier i2c_status.h

Sur le programme principal on commence par l’appel des bibliothèques que nous utilisons comme ‘avr/io.h’ qui nous permet d’utiliser les ports et les registres en écrivant leurs noms au lieu d’utiliser les adresses. Ainsi on fait appel à la bibliothèque ‘avr/interrupt.h’ qui nous permet d’utiliser les interruptions du microcontrôleur. On fait appel à la bibliothèque que nous avons créée ("i2c_status.h") pour utiliser les noms des codes et les messages. Deux autres bibliothèques que nous avons utilisées pour nous permettre l’utilisation de certains types de variables.

Ensuite, on définit certaines fonctions que nous utilisons dans notre main, une fonction d’initialisation des ports, nous utilisons le protocole de communication série UART donc il faut

mettre TX en sortie et RX en entré. Prochainement, nous allons utiliser plus de port pour pouvoir prendre des mesures réel à l'aide des capteurs. On rappelle que le protocole de communication série est utilisé seulement pour qu'on puisse voir l'état de bus I2C, et ça ne fait pas partie de notre projet, c'est pour cela on trouve la condition suivant lors de l'initialisation ou de l'utilisation du bus série :

```
#if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0
.
.
.
#endif
```

Où `AVR_DEBUG_TWI` est un flag qu'on définit au début du code et on le donne 1 si on veut l'utiliser, sinon on le donne 0 pour ne pas l'utiliser. Cette méthode permet de gagner le temps au lieu d'aller mettre en commentaire ou de retirer tous les parties où nous utilisons ce protocole de communication.

Autre fonction que nous avons créée s'appelle `TWI_SLAVE_INIT ()`, cette dernier permet l'initialisation du protocole de communication I2C, on active l'I2C par mettre la valeur hexadécimale 0x04 sur le registre de contrôle avant, ensuite on met l'adresse sur le registre TWAR qui est le registre d'adresse, et après nous initialisant l'état du registre control qui contient certain bits importants pour que l'I2C fonctionne, ces bits sont :

- **TWINT (TWI Interrupt Flag)** : ce bit est défini (à l'état bas) par le matériel lorsque l'I2C a terminé son travail en cours, et attend une réponse du logiciel. Ce bit génère alors une interruption lorsqu'il change d'état. Il faut le réinitialisé à la fin de la routine d'interruption on le met à l'état logique '1'.
- **TWEN (TWI Enable)** : c'est le bit qui permet d'activer l'I2C et son interface.
- **TWEA (TWI Enable Acknowledge)** : ce bit permet le contrôle de la génération de l'impulsion d'acquiescement s'il est définis à '1'.
- **TWIE (TWI Interrupt Enable)** : Lorsque ce bit est écrit sur un et que le bit 'I' dans SREG est mis à '1', la demande d'interruption TWI sera activée tant que l'indicateur TWINT est haut.

Ces quatre bits sont les plus utilisé dans notre code.

On arrive à la routine d'interruption où on utilise un registre qu'on doit le définir avant de commencer l'explication de ce que nous avons mis dans le code. Ce registre s'appelle **TWSR (TWI Status Register)**, c'est sur ce dernier où les codes de statuts de bus que nous avons

définie précédemment seront stocké, donc à chaque fois une routine d'interruption est générée, on doit comparer ce registre avec les codes qu'on a sur le fichier header. Notons que les 3 bits de poids faible de ce registre n'entre pas dans la définition des codes c'est pour cela à chaque fois on compare le code reçu et pour assurer que ces bits ne nous perturbent pas, on met l'instruction suivante :

```
if((TWSR & 0xF8) == SR_SLA_ACK)
```

À l'intérieur de 'if ' on trouve (TWSR & 0xF8) qui s'exécute avant de la comparaison. Cette instruction est une fonction logique AND qui nous permet de garder les bits 7 à 4 et de mettre à zéro les bits qui restent. On trouve plusieurs codes mais ce qui nous intéresse sont les suivants :

- *SR_SLA_ACK* : l'interface I2C a détecté son adresse en mode écriture et il a renvoyé un ACK.
- *SR_DATA_ACK* : Réception de donnée envoyée par le master lors de l'adressage avec l'adresse du Slave. On trouve le mot envoyé dans le registre TWDR (TWI Data Register). ACK a été envoyé par le Slave.
- *SR_DATA_NACK* : Même définition de *SR_DATA_ACK* mais cette fois-ci Non ACK retourné.
- *ST_SLA_ACK* : l'interface I2C a détecté son adresse en mode lecture (le master demande des données) et ACK retourné. C'est sur cette phase où on doit préparer la première donnée à envoyer s'il y'en a plusieurs en le mettant dans TWDR.
- *ST_DATA_ACK* : le mot chargé dans le TWDR a été envoyé et ACK a été détecté. Ici en charge les données qui restent à envoyer.
- *TW_STOP* : un bit de stop est détecté sur le bus, fin de la communication.

Dans la fonction main, on commence par l'appel aux fonctions d'initialisation des ports, de l'UART et de l'I2C, ensuite on active le bit d'interruption générale par l'instruction sei (). Après on crée une boucle infinie en utilisant while(1) et à l'intérieur on ne fait rien que lorsque le code envoyé par l'Arduino est 'M' c'est-à-dire faire les mesures où on commence l'incréméntation en assurant l'état des mesures si c'est fini la réponse sera 'R', si ce n'est pas prêt on charge 'N' au cas où l'arduino demande l'état des mesures lorsque le Slave entraîne de les effectuer.

On finit cette partie par des captures de l'environnement de communication virtuel PuTTY qui nous permet de voir le déroulement de l'échange du côté de la carte Olimex, le scénario utilisé est le scénario 4.

```
address Receive with SLA_ACK IN Write Mode

Data received with SLA_ACK addressable and Code is :
H
A Stop condition arrived

address Receive with SLA_ACK IN READ Mode and DATA has been charged in TWDR
Data has been TRANSMETED and NOT ACK has been received

address Receive with SLA_ACK IN Write Mode

Data received with SLA_ACK addressable and Code is :
M
A Stop condition arrived
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
address Receive with SLA_ACK IN Write Mode

Data received with SLA_ACK addressable and Code is :
S
A Stop condition arrived

address Receive with SLA_ACK IN READ Mode and DATA has been charged in TWDR
Data has been TRANSMETED and NOT ACK has been received

address Receive with SLA_ACK IN Write Mode

Data received with SLA_ACK addressable and Code is :
G
A Stop condition arrived

address Receive with SLA_ACK IN READ Mode and DATA has been charged in TWDR

Data has been TRANSMETED with SLA_ACK addressable
Data has been TRANSMETED with SLA_ACK addressable
Data has been TRANSMETED with SLA_ACK addressable
Data has been TRANSMETED with SLA_ACK addressable
Data has been TRANSMETED with SLA_ACK addressable
```

Détection d'adresse en mode écriture

Réception du code 'H'

Détection d'adresse en mode lecture et la réponse chargé dans le TWDR

Réception du code 'M' est départ d'incréméntation

Réception du code 'S' : Arduino demande l'état des mesures

Chargement de la réponse et ensuite envoyer : 'R'

Réception du code 'G' Arduino demande les données


```
address Receive with SLA_ACK IN READ Mode and DATA has been charged in TWDR
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED with SLA_ACK addressable
```

```
Data has been TRANSMETED and NOT ACK has been received
```

Envoi des données

NACK : dernier élément envoyé

VIII. Conclusion

En conclusion, on retiendra tout au long de ce projet qu'il existe deux protocoles de communication largement utilisés dans l'industrie embarquée et aéronautique pour faire communiquer deux charges d'une manière ou d'une autre : SPI & I2C. Nous sommes tout à fait satisfaits des résultats obtenus et objectifs atteints durant cet apprentissage pédagogique. En effet, nous avons appris à maîtriser ces protocoles de communications évoqués plus haut pour faire communiquer deux cartes électroniques par des codes à l'aide de différents logiciels, ce qui nous a permis d'approfondir nos connaissances en programmation. Par ailleurs, nous

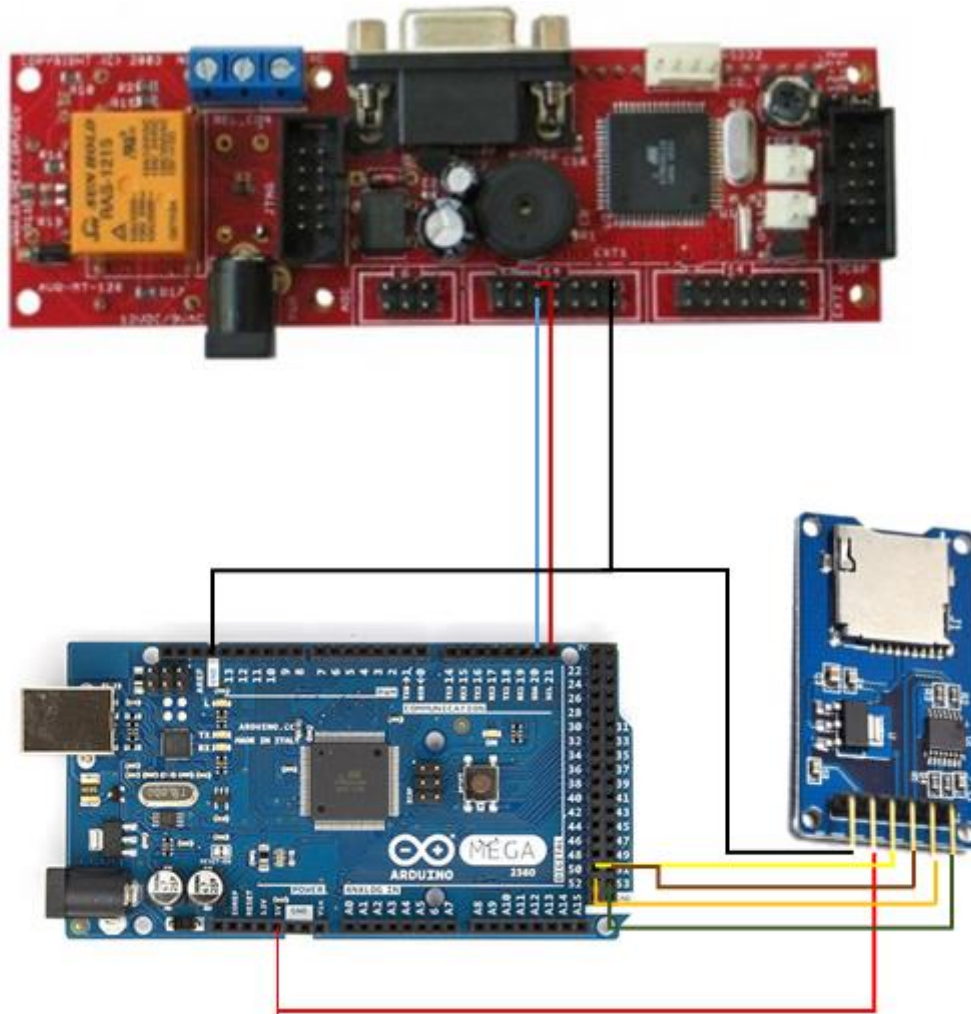
sommes convaincus que le travail élaboré n'est qu'une étape primaire aussi bien pour nos carrières professionnelles envisagées que pour des études approfondies.

Annexes

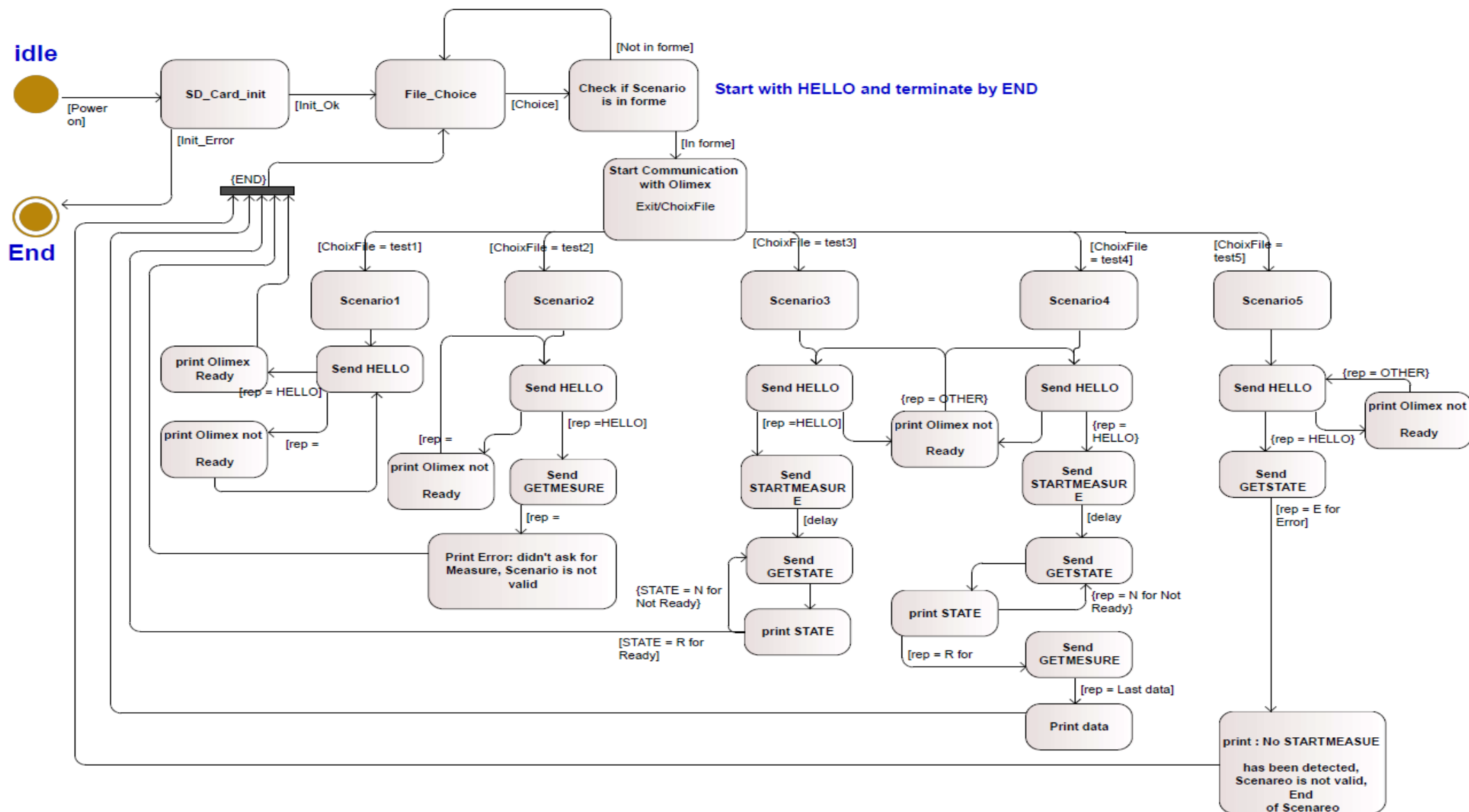
Annexe 1 : Caractéristique de l'ATmega128A

- Architecture RISC avancée - 133 instructions puissantes
- 32 registres de fonctionnement à usage général
- Fonctionnement entièrement statique - Jusqu'à 16 MIPS Débit à 16 MHz
- 128 Ko de mémoire de programme Flash auto-programmable
- EEPROM 4 Koctets
- SRAM interne 4 Koctets
- Interface de communication série SPI (Master-Slave)
- Interface série à deux fils orientée octets I2C
- Double USART série programmable
- 2 Timer/compteur de 8 bits avec Prescalers et Compare Modes séparé
- 2 Timer/compteur de 16 bits Prescalers, Compare Modes et capture Mode séparé
- 2 channels de signal PWM de 8 bits
- 6 channels de signal PWM avec résolution programmable de 2 à 16 bits
- 8 channels de convertisseur analogique numérique 10 bits
- Sources d'interruption externes et internes
- E / S et packages
 - 53 E / S programmable
- Tensions de fonctionnement
 - 2.7 - 5.5V
- Fréquence de fonctionnement
 - 0 - 16MHz

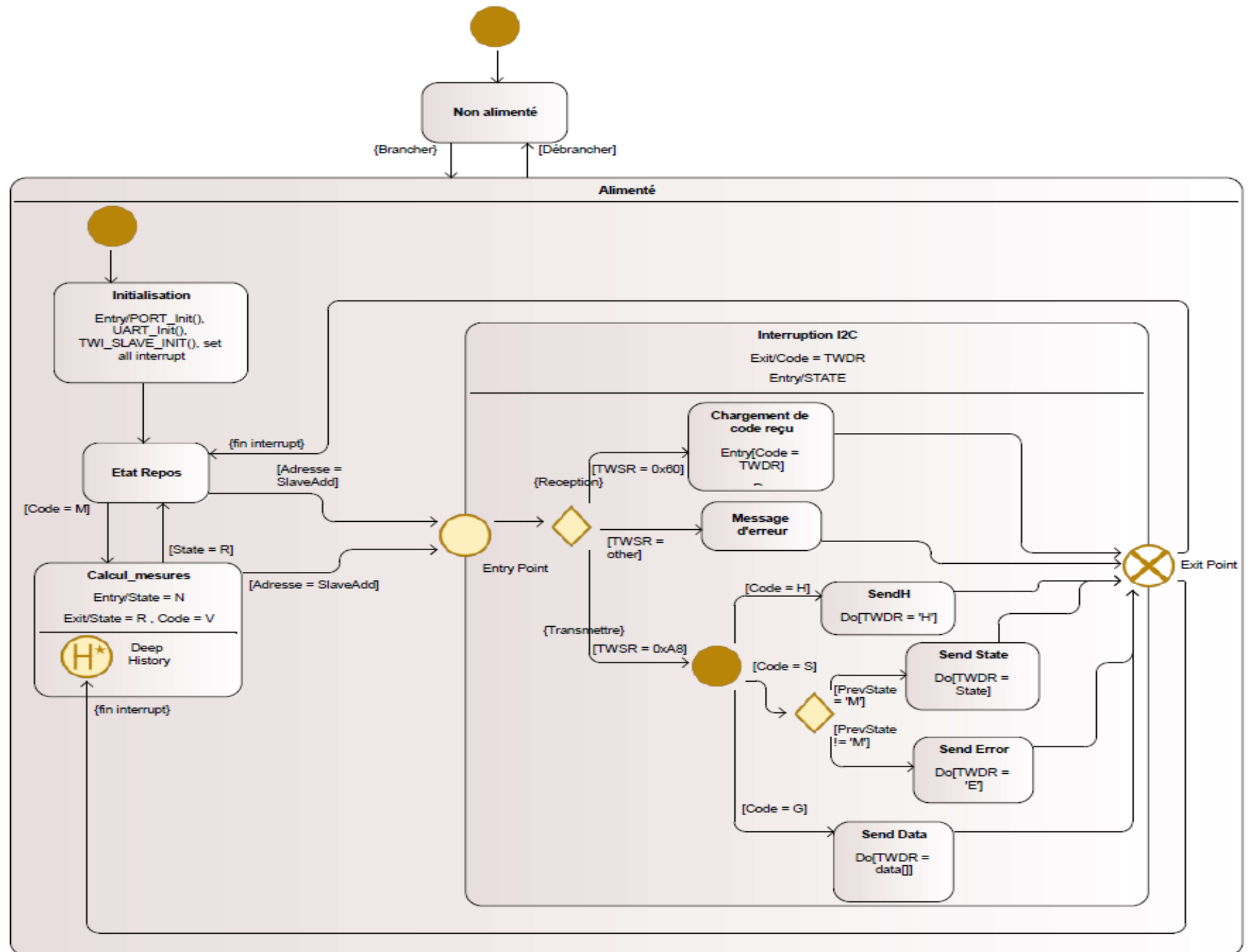
Annexe 2 : Câblage entre les différentes cartes



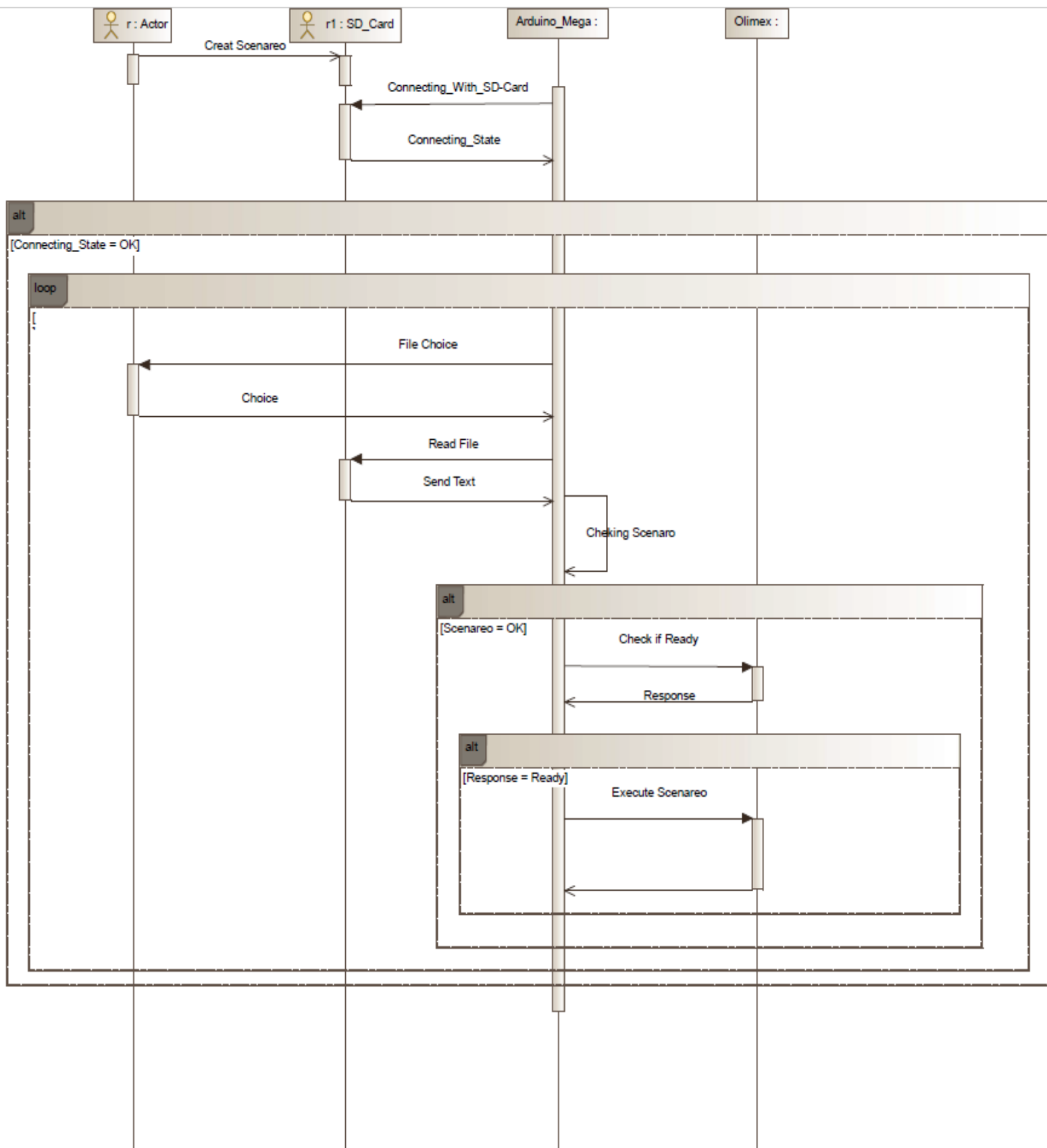
Annexe 3 : Diagramme d'état coté Arduino (OBC)



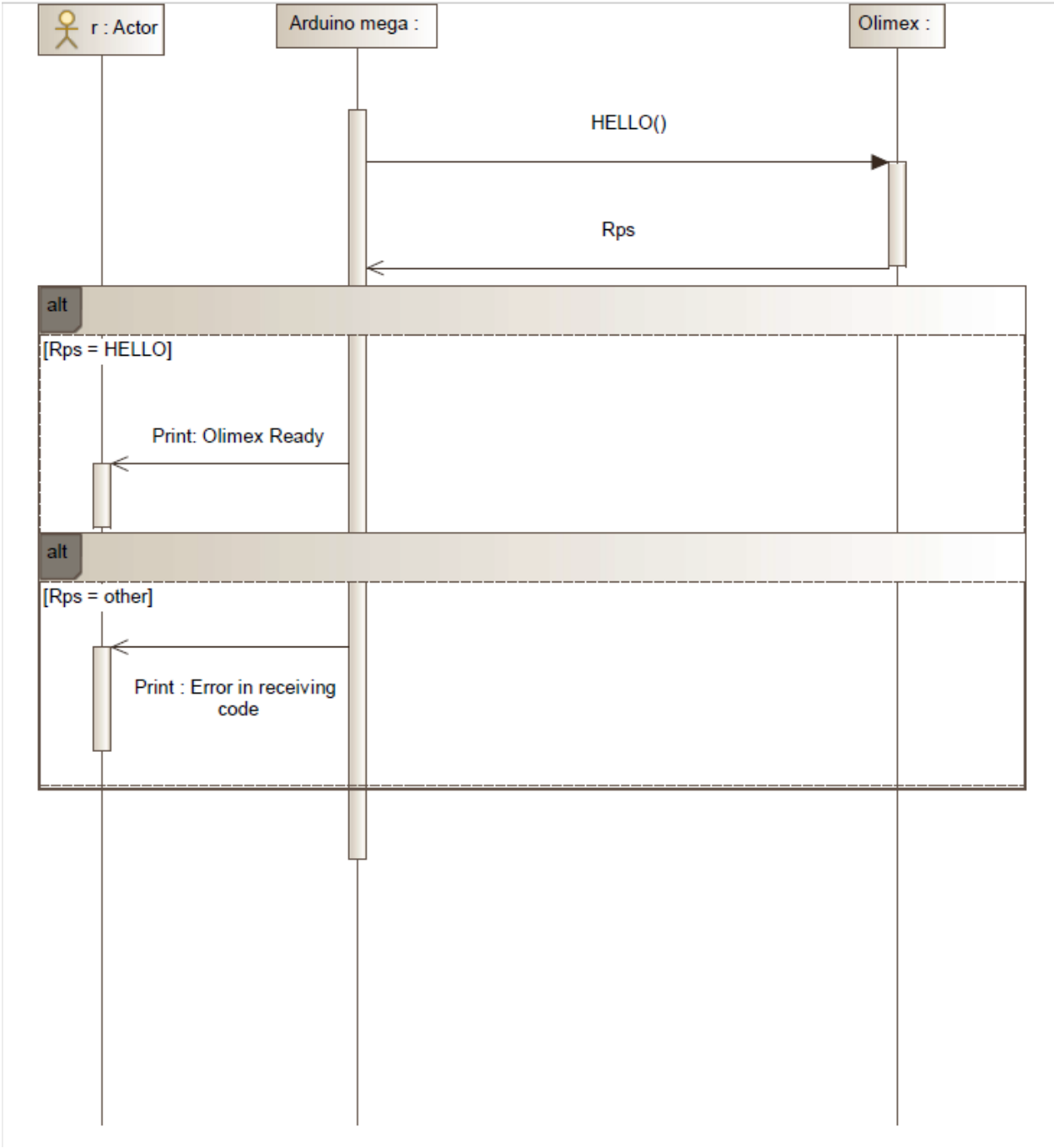
Annexe 4 : Diagramme d'état coté Olimex (EDMON)



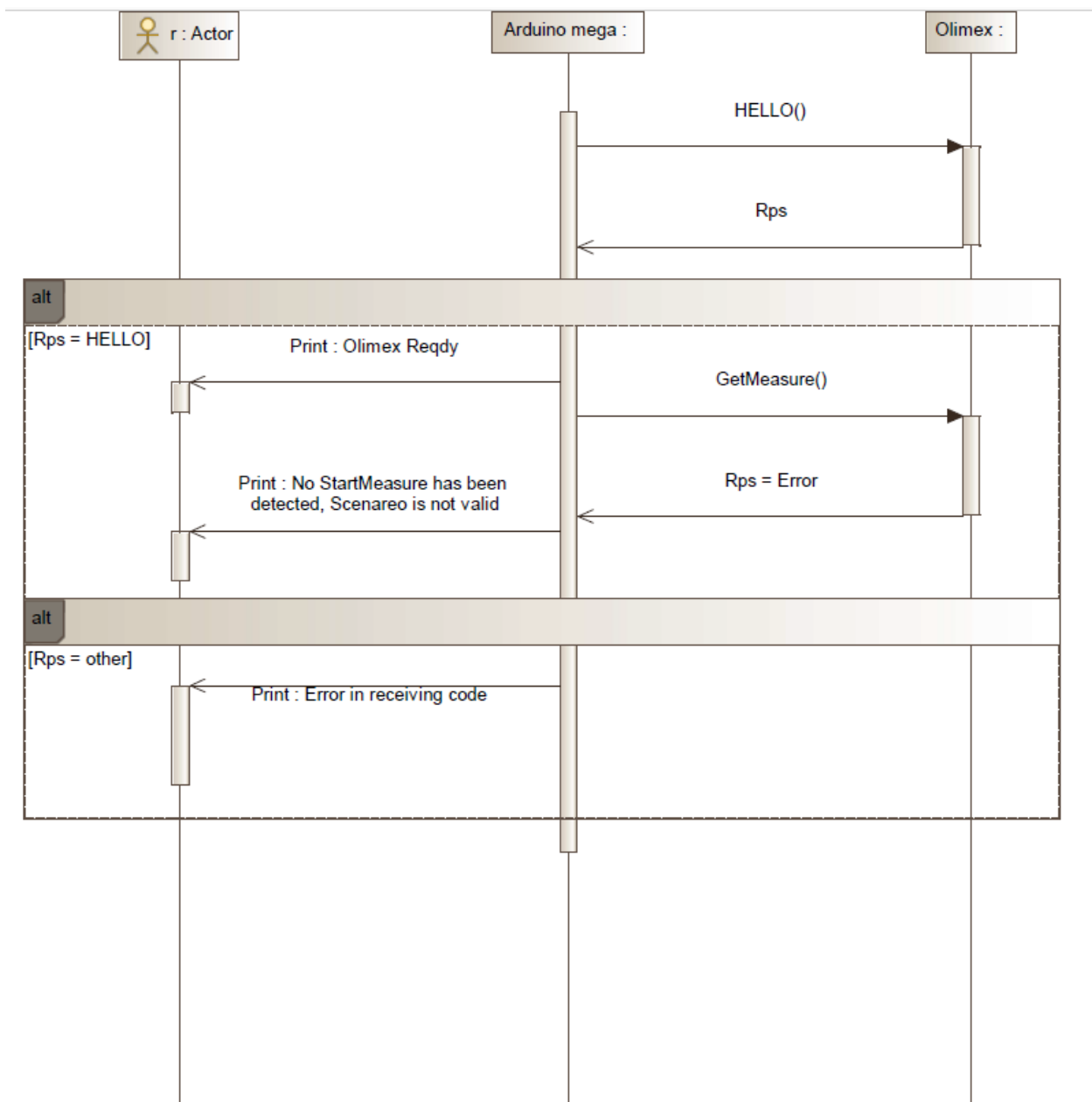
Annexe 5 : Diagramme de séquence principale – Interaction entre acteur



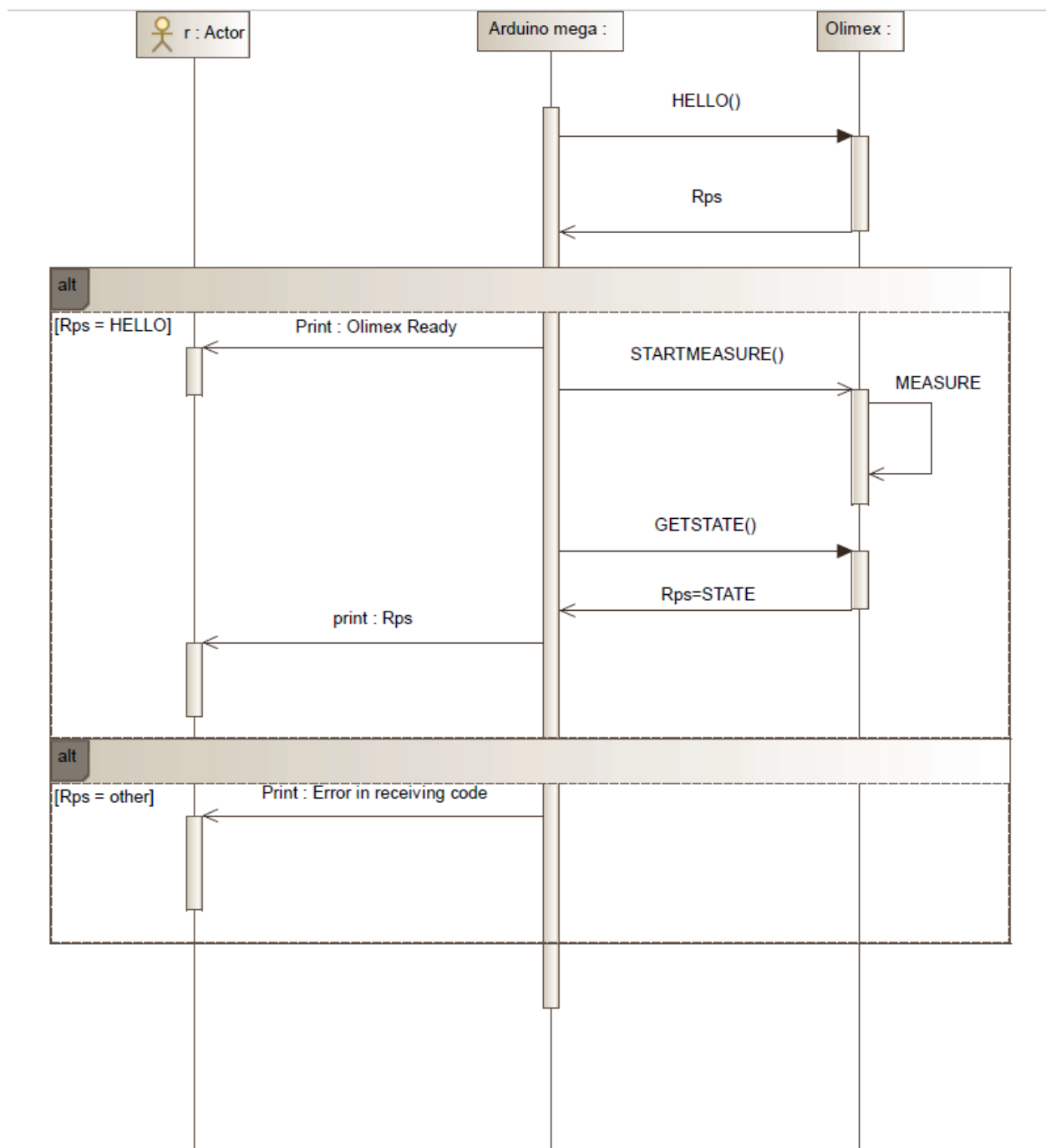
Annexe 6 : Diagramme de séquence secondaire – Scénario 1



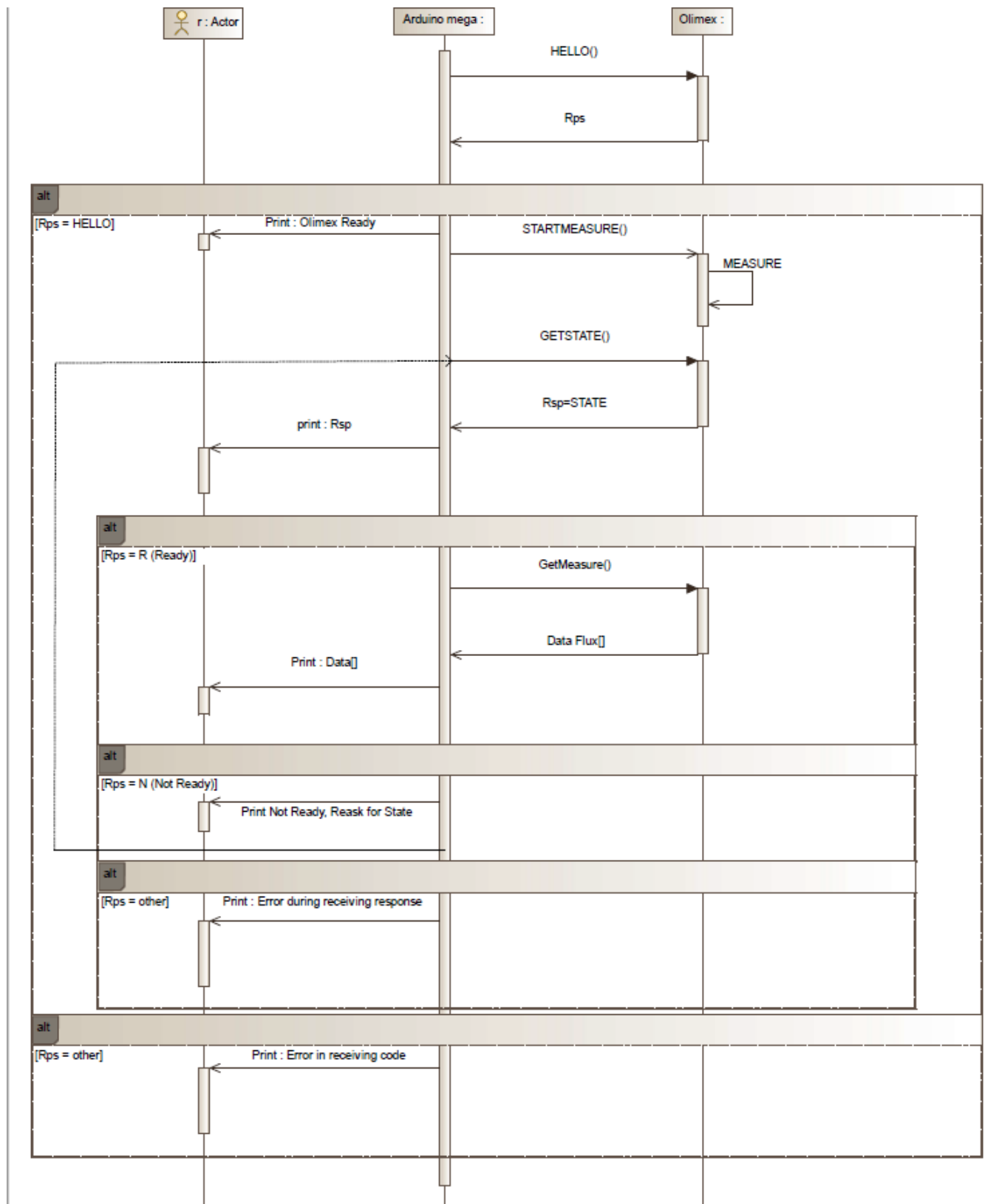
Annexe 7 : Diagramme de séquence secondaire – Scénario 2



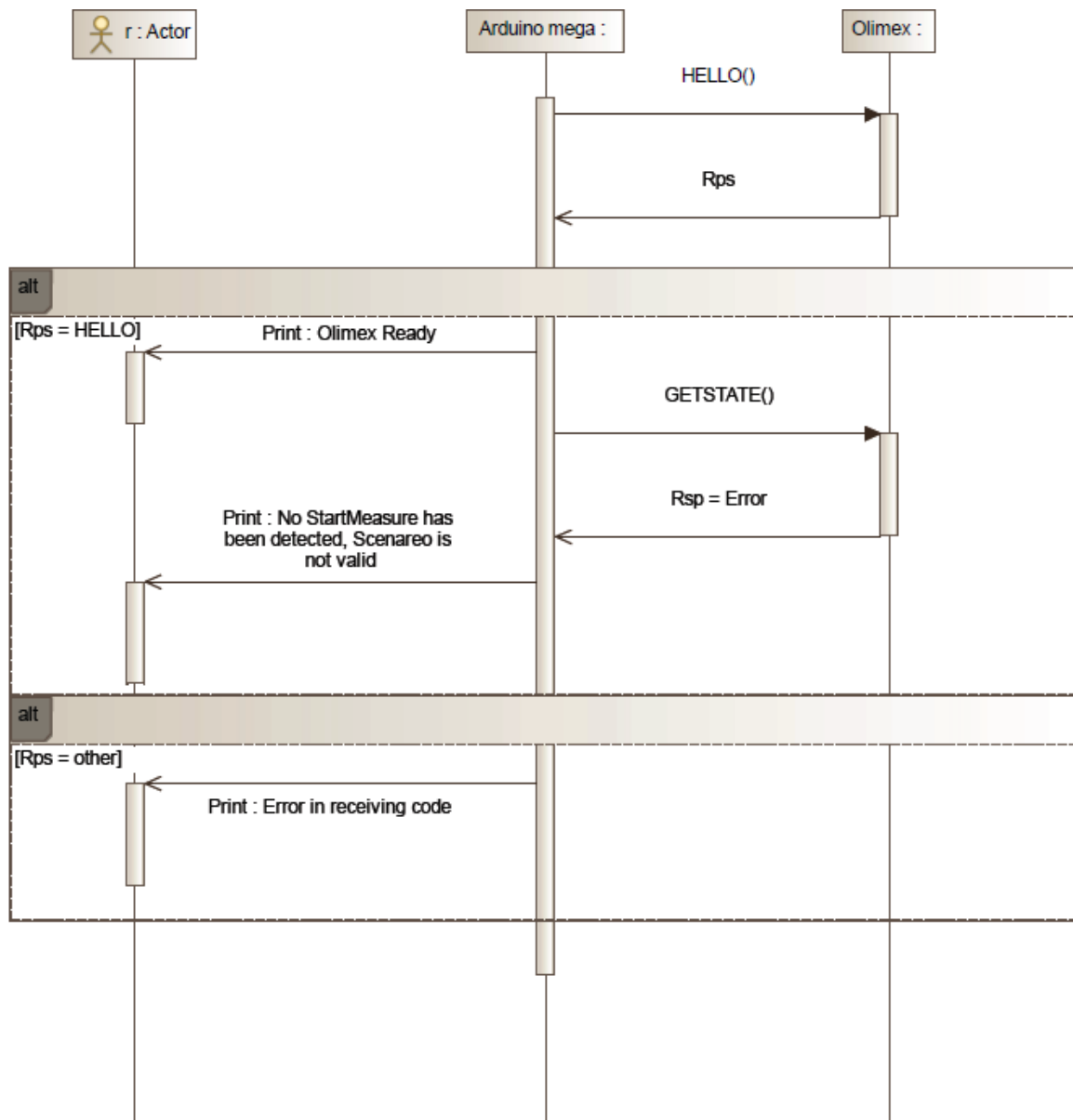
Annexe 8 : Diagramme de séquence secondaire – Scénario 3



Annexe 9 : Diagramme de séquence secondaire – Scénario 4



Annexe 10 : Diagramme de séquence secondaire – Scénario 5



Annexe 11 : Programme d'Arduino mega

```
/*
 * POSSIBLE CODES FOR COMMUNICATION WITH OLIMEX
 // code = 'H' => One check => response from Slave is 'H' if it is ready
 // code = 'M' => Start Measure => No response from Slave
 // code = 'S' => Get state => Response is 'N' if Not Ready
 //                                     or 'R' if Ready
 // code = 'G' => Get Measure => Response is a flux of data
 */
#include <Wire.h>
#include <SD.h>
#include <SPI.h>
// declaration des variables pour la communication avec la carte Olimex
#define SLAVE_ADDR 9 // Define Slave Address
#define sizeofdata 20 // Size of the data that will receive
byte data[50] = {0}; // 50 memory slots reserved for received data
char Resp = 'V'; // will contain the respons
char STATE = 'N'; // will contain the state of measure

// variables pour la communication avec la carte memoire
File myFile;
int CS_SD = 53;
int tabsize = 5;
String NomFiles[] = {"test1.txt", "test2.txt", "test3.txt", // les noms des fichiers représentant les scénarios
                    "test4.txt", "test5.txt"};
String Scenareo[] = {"HELLO\nEND", "HELLO\nGETMESURE\nEND", // 5 scénaios qui seront dans les ficheirs texts
                    "HELLO\nSTARTMESURE\nGETSTATE\nEND",
                    "HELLO\nSTARTMESURE\nGETSTATE\nGETMESURE\nEND",
                    "HELLO\nGETSTATE\nGETMESURE\nEND"};
String *line = (String*)calloc(tabsize, sizeof(String)); // tableau qui stock les lignes lu dans un fichier
String file = ""; // variable qui stock le choix du fichier selectionné par l'utilisateur
enum etats {Wait, SendH, SendM, SendS, SendG};
etats etat = Wait;

void setup() {
    Wire.begin(); // start I2C Bus as Master:
    Serial.begin(9600); // initialisation du moniteur serie
    pinMode(CS_SD, OUTPUT); // SD card Chip Select for SPI bus
    SDinitialization();
    for(int i = 0 ; i<5 ; i++){ // 5 represente le nombre de fichier, à changer si le nombre de fich
        checkIfexists(NomFiles[i]); // fonction qui permet de voir si un fichier exist, si oui il sera s
        writeToFile(NomFiles[i], Scenareo[i]); // écrire dans le fichier[i] le scenareo[i]
    }
    Serial.println("\t\tI2C Master Demonstration");
}

void SDinitialization(){ // Fonction pour l'initialisation de la carte memoire
    Serial.print("Initializing SD card...");
    if (SD.begin())
    {
        Serial.println("SD card is ready to use.");
    } else
    {
        Serial.println("SD card initialization failed");
        return;
    }
}
```

Annexe 11 : Programme d'Arduino mega

```
String ChoixFile(){ // Fonction qui permet de choisir un fichier
    int choixF;
    Serial.println("Choice the scenareo that you want to execute :");
    indice :
    Serial.println("1 : test1.txt\n2 : test2.txt\n3 : test3.txt\n4 : test4.txt\n5 : test5.txt");
    char inv = Serial.read(); // pour eviter le bruit de l espace apres lecture
    while(Serial.available() == 0);
    choixF = (int) (Serial.read())-48;// retransche 48 du code ascii pour la mise en forme du valeur

    if(choixF < 1 || choixF > 5){
        Serial.println("invalide choice...Please make a new choice : ");
        goto indice;
    } else {
        Serial.print("Your choice is : ");
        Serial.println(NomFiles[choixF-1]);
    }
    return NomFiles[choixF-1];
}

void checkIfexists(String file){ // Fonction qui véifier si un fichier existe
    if(SD.exists(file)){
        // Serial.println("the file exists");
        SD.remove(file); // si le fichier existe on le supprime pour réécrire de nouveau
        if(SD.exists(file)){
            Serial.println("the file still exists");
        }
    } else{
        Serial.println("the file was replaced");
    }
}

void writeToFile(String file, String text){ // Fonction permettant d'écrire dans un fichier
    myFile = SD.open(file, FILE_WRITE);
    if (myFile){
        myFile.println(text);
        Serial.print("Writing to ");
        Serial.print(file);
        Serial.println(" is Done.");
    } else{
        Serial.println("Couldn't write to file");
    }
    myFile.close();
}

int readText(String file,String *tab, int tabsz){ // Fonction qui permet de lire un fichier
    myFile = SD.open(file, FILE_READ);
    String received = "";
    char ch;
    int i=0;

    while(myFile.available()){
        ch = myFile.read();
        if(ch == '\n'){
            tab[i] = received;
            received = "";
            if(i > tabsz){
                Serial.println("i indexe passed size of table");
                goto fin;
            }
            i += 1;
            //Serial.println(i);
        }
        else{
            received += ch;
        }
    }
    fin :
    myFile.close();
    return i; // on retourne le nombre de lignes qui sont dans le fichier
}
```

Annexe 11 : Programme d'Arduino mega

```
void loop() {
  start :
  file = ChoixFile();      // fonction permet de choisir un des fichiers
  int nbrline = readText(file,line,tabsz);
  Serial.println("Scenareo is : ");
  for(int i = 0 ; i < nbrline ; i++){
    Serial.println(line[i]);
  }

  if (line[0] == "HELLO"){
    Serial.println("Scenareo is Ok...Begin communication with Olimex");
    int l = 0;
    for(int i = 0 ; i < nbrline ; i++){
      if(line[i] == "HELLO") etat = SendH;
      else if(line[i] == "STARTMESURE") etat = SendM;
      else if(line[i] == "GETSTATE") etat = SendS;
      else if(line[i] == "GETMESURE") etat = SendG;
      else etat = Wait;

      switch(etat){
        case SendH :
          // Print to Serial monitor
          Serial.println("Begin Transmission of 'H' code");
          reask :
          // Start transfer of the 'H' code
          Wire.beginTransmission(SLAVE_ADDR); // start communication with slave
          Wire.write('H');                    // send the code byte
          Wire.endTransmission();             // end of Transmission

          delay(40); // wait for 40ms to assure that the bus is available

          Serial.println("End of Transmission and Begin Receive Response");
          // Receive the response from the slave
          Wire.requestFrom(SLAVE_ADDR,1);    // demand for a request
          while (Wire.available()){
            Resp = Wire.read();              // Read the response from Slave
            Serial.print("End of Receive and the Response is : ");
            Serial.println(Resp);
          }

          if(Resp == 'H') Serial.println("Olimex is Ready for commication");
          else {
            Serial.println("Olimex is not Ready..Retransmission of H");
            goto reask;
          }
          break;

        case SendM:
          // Print to Serial monitor
          Serial.println("Begin Transmission of 'M' code");
          // Start transfer of the code
          Wire.beginTransmission(SLAVE_ADDR); // start communication with slave
          Wire.write('M');                    // send the code byte
          Wire.endTransmission();             // end of Transmission

          delay(40); // wait for 40ms to assure that the bus is available
          Serial.println("Transmission of 'M' code is done");
          delay(200); // wait for 200ms to ensure that slave finished the measure
          break;
      }
    }
  }
}
```

Annexe 11 : Programme d'Arduino mega

```
case SendS:
    ReAskForMeasur:
        // Print to Serial monitor
        Serial.println("Begin Transmission of 'S' code");
        // Start transfer of the code
        Wire.beginTransmission(SLAVE_ADDR); // start communication with slave
        Wire.write('S');                    // send the code byte
        Wire.endTransmission();             // end of Transmission

        delay(40);

        Serial.println("Receive Response :");
        Wire.requestFrom(SLAVE_ADDR,1);    // demand for a request of the state
        while (Wire.available()){
            Resp = Wire.read();              // Read the response from Slave
        }

        if (Resp == 'N'){
            Serial.println("Measure is not Ready");
            Serial.println("Re-asking for the state of measure");
            goto ReAskForMeasur;
        }
        else if (Resp == 'R'){
            Serial.println("Measure is Ready");
            STATE = 'R';
        }

        else if (Resp == 'E'){
            Serial.println("Error message has been received..Scenareo is not valid..may ask Olimex to start mesure");
            goto start;
        }
        else{
            Serial.println("Response is not valid..check Olimex code");
            goto start;
        }
        break;

case SendG:
    if (STATE == 'R'){
        STATE == 'N'; // initialise STATE
        // start of transfer the code byte
        Serial.println("Begin Transmission of 'G' code");
        Wire.beginTransmission(SLAVE_ADDR); // start communication with slave
        Wire.write('G');                    // send the code byte
        Wire.endTransmission();             // end of Transmission

        delay(40);

        // Receive the data
        Serial.println("End of Transmission and Begin Receiving data");
        Wire.requestFrom(SLAVE_ADDR,sizeofdata);
        byte i = 0;
        while(Wire.available()){
            data[i] = Wire.read();
            if(data[i] == 'E'){
                Serial.println("Error message has been received..Scenareo is not valid..may ask for State before");
                goto start;
            }
            Serial.println(data[i]);
            i++;
        }
        Serial.println("All Measure request has been received");
    }
    else Serial.println("Didn't ask for Mesure..Scenareo is not valid");
    break;
}
}
}
```


Annexe 12 : Programme du microcontrôleur ATmega128A

```
#ifndef F_CPU
#define F_CPU 16000000UL
#endif

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <string.h>
#include <stdint.h>

#include "i2c_status.h"

#define __AVR_ATMEGA128__ 1
#define OSCSPEED 16000000 // IN Hz
#define SLAVE_ADDR 0x09 // SLAVE address

// DEBUG
// @brief STOP DEBUG ==> AVR_DEBUG_TWI 0
// @brief ENABLE DEBUG ==> AVR_DEBUG_TWI 1
#define AVR_DEBUG_TWI 1

unsigned char CODE = 'N'; // CODE FOR SEEING WHAT THE MASTER LOOKING FOR
unsigned char PrevC = 'N'; // Previous code used to check if the sequence was in form
uint8_t DATA[50] = {0}; // THE DATA WILL BE IN THIS BYTES NAMED DATA
unsigned char STATE = 'N'; // THIS VARIABLE WILL HAVE THE STATE OF MESURE ; READY (R)/NOT YET (N)
int CWPt = 0; // THIS INTEGER ARE USED TO COUNT HOW MATCH DATA THERE ARE
int j = 0; // pointer of the data cases

void PORT_Init()
{
    PORTA = 0b00000000; DDRA = 0b01000000; //Relay set as output (Bit6 = 1)
    PORTB = 0b00000000; DDRB = 0b00000000;
    PORTC = 0b00000000; DDRC = 0b11101111; //LCD connection with PORTC
    PORTD = 0b11000000; DDRD = 0b00000100; //TX set as output (Bit3 = 1)
    PORTE = 0b00000000; DORE = 0b00110000; //Buzzer set as output (Bit4 = 1, Bit5 = 1)
    PORTF = 0b00000000; DDRF = 0b00000000;
    PORTG = 0b00000000; DD RG = 0b00000000;
}

#if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0
void UART_Init(uint32_t Baud)
{
    unsigned int BaudRate = OSCSPEED / (16 * Baud) - 1; /* as per pg. 173 of the user manual */

    //set BaudRate to registers UBRR1H and UBRR1L
    UBRR1H = (unsigned char) (BaudRate>>8);
    UBRR1L = (unsigned char) BaudRate;

    UCSR1B = UCSR1B | 0b00011000; //enable Receiver and Transmitter (Bit3 = 1, Bit4 = 1)
    UCSR1C = UCSR1C | 0b1000110; //Set frame format: 8 data (Bit1 = 1, Bit2 = 1), 1 stop bit (Bit3 = 0)
}

void UART_Transmit(unsigned char data)
{
    while (!(UCSR1A & 0b00100000)); //waiting until buffer is ready to receive

    UDR1 = data;
}
#endif

void TWI_SLAVE_INIT()
{
    cli(); // arreter tout interruption; desable general interrupt bit 'I' of SREG
    TWCR = 0x04; // ENABLE TWI
    TWAR = (SLAVE_ADDR << 1); // address shifted a gauche d'un pas pour qu'il soit sur la bonne position
    TWCR = (1<<TWINT) | (1<<TWEA) | (1<<TWEN) | (1<<TWIE); // TWINT Cleared par la mise a 1
    // TWEN : enable i2c
    // TWEA : enable acknowledge
    // TWIE : enable i2c interruption
}

ISR(TWI_vect) // Vecteur d'interruption de bus i2c
{
    /****** Slave Receiver Mode *****/

    // SLA+W received, ACK returned : 0x00
    if((TWSR & 0xF8) == SR_SLA_ACK) {
        #if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0
            for (unsigned char i = 0 ; i <= strlen(MSG1) ; i++)
            {
                UART_Transmit(MSG1[i]);
            }
        #endif

        TWCR |= (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN); // Clear TWINT and prepare to receive new byte
    }

    // Arbitration lost in SLA+RW / SLA+W received, ACK returned : 0x08
    else if ((TWSR & 0xF8) == SR_ARB_LOST_SLA_ACK)
    {
        #if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0
            for (unsigned char i = 0 ; i <= strlen(MSG2) ; i++)
            {
                UART_Transmit(MSG2[i]);
            }
        #endif

        TWCR |= (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN);
    }
}
```

Annexe 12 : Programme du microcontrôleur ATmega128A

```
// general call received, ACK returned : 0x70
else if ((TISR & 0xF8) == SR_GEN_CALL_ACK)
{
    TWCR |= (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN);

    #if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0
        for (unsigned char i = 0 ; i <= strlen(MSG3) ; i++)
        {
            UART_Transmit(MSG3[i]);
        }
    #endif
}

// Code received, ACK returned : 0x80 ; Previously addressed with own SLA+W
else if ((TISR & 0xF8) == SR_DATA_ACK)
{
    CODE = TWDR;

    #if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0
        for (unsigned char i = 0 ; i <= strlen(MSG4) ; i++)
        {
            UART_Transmit(MSG4[i]);
        }

        UART_Transmit(CODE);
    #endif

    TWCR |= (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN);
}
```

```
// data received, NACK returned : 0x88
else if ((TISR & 0xF8) == SR_DATA_NACK)
{
    CODE = TWDR;
    #if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0
        for (unsigned char i = 0 ; i <= strlen(MSG5) ; i++)
        {
            UART_Transmit(MSG5[i]);
        }
        UART_Transmit(CODE);
    #endif

    TWCR |= (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN);
}

// general call data received, ACK returned : 0x90
else if ((TISR & 0xF8) == SR_GEN_CALL_DATA_ACK)
{
    CODE = TWDR;
    #if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0
        for (unsigned char i = 0 ; i <= strlen(MSG6) ; i++)
        {
            UART_Transmit(MSG6[i]);
        }
        UART_Transmit(CODE);
    #endif

    TWCR |= (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN);
}
```

```
// general call data received, NACK returned : 0x98
else if ((TISR & 0xF8) == SR_GEN_CALL_DATA_NACK)
{
    CODE = TWDR;
    #if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0
        for (unsigned char i = 0 ; i <= strlen(MSG7) ; i++)
        {
            UART_Transmit(MSG7[i]);
        }
        UART_Transmit(CODE);
    #endif

    TWCR |= (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN);
}
```

```
/****** Slave Transmitter Mode *****/
// SLA+R Received and ACK returned : 0xA8
else if ((TISR & 0xF8) == ST_SLA_ACK) // MASTER DEMANDE A DATA, SO WE NEED TO PREPARE IT HERE
{
    switch (CODE){
        case 'H' : TWDR = 'H'; break;
        case 'S' :
            if (PrevC == 'M') // check if the previous message was StartMeasure then send the State
            {
                TWDR = STATE;
            } else {TWDR = 'E';} // if not, send E for Error
            PrevC = 'S'; // Previous code is become 'S' for GetState
            break;
        case 'G' :
            CHPT -- 1; // decrement CHPT by 1
            TWDR = DATA[j]; // charging the first data in TWDR; j = 0 evry time data is charged
            j ++ 1; // increment j by 1 to point on the next case of data
            break;
        default : TWDR = 'E'; break; // send E for Error
    }

    #if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0
        for (unsigned char i = 0 ; i <= strlen(MSG9) ; i++)
        {
            UART_Transmit(MSG9[i]);
        }
    #endif

    TWCR |= (1<<TWEA) | (1<<TWEN);
    //TWCR |= (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN);
}
```

Annexe 12 : Programme du microcontrôleur ATmega128A

```
// Arbitration lost in SLA+R/W; SLA+W has been received; ACK has been returned : 0xB0
else if ((TWSR & 0xF8) == ST_ARB_LOST_SLA_ACK)
{
    #if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0
        for (unsigned char i = 0; i <= strlen(MSG2); i++)
        {
            UART_Transmit(MSG2[i]);
        }
    #endif
    TWCR |= (1<<TWEA) | (1<<TWEN);
    //TWCR |= (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN);
}

// Data byte in TWDR has been transmitted; ACK has been received : 0xB8
else if ((TWSR & 0xF8) == ST_DATA_ACK)
{
    if (CHPT == 0)
    {
        TWDR = 'L';          // charge TWDR by 'L' to tell master that was the last data
        j = 0;              // initiate the pointer
        CHPT = 0;

        #if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0
            for (unsigned char i = 0; i <= strlen(MSG12); i++)
            {
                UART_Transmit(MSG12[i]);
            }
        #endif
    }

    else{
        CHPT -= 1;          // decrement CHPT by 1
        TWDR = DATA[j];    // charging the second data in TWDR
        j += 1;            // increment j by 1 to point on the next case of data

        #if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0
            for (unsigned char i = 0; i <= strlen(MSG11); i++)
            {
                UART_Transmit(MSG11[i]);
            }
        #endif
    }

    TWCR |= (1<<TWEA) | (1<<TWEN);
    // TWCR |= (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN);
}

// Data byte in TWDR has been transmitted; NOT ACK has been received : 0xC0
else if ((TWSR & 0xF8) == ST_DATA_NACK)
{
    #if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0
        for (unsigned char i = 0; i <= strlen(MSG13); i++)
        {
            UART_Transmit(MSG13[i]);
        }
    #endif
    TWCR |= (1<<TWEA) | (1<<TWEN);
    // TWCR |= (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN);
}

// Data byte in TWDR has been transmitted (TWEA = "0"); ACK has been received : 0xC8
else if ((TWSR & 0xF8) == ST_DATA_ACK_TWEA0)
{
    #if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0
        for (unsigned char i = 0; i <= strlen(MSG14); i++)
        {
            UART_Transmit(MSG14[i]);
        }
    #endif
    TWCR |= (1<<TWEA) | (1<<TWEN);
    // TWCR |= (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN);
}

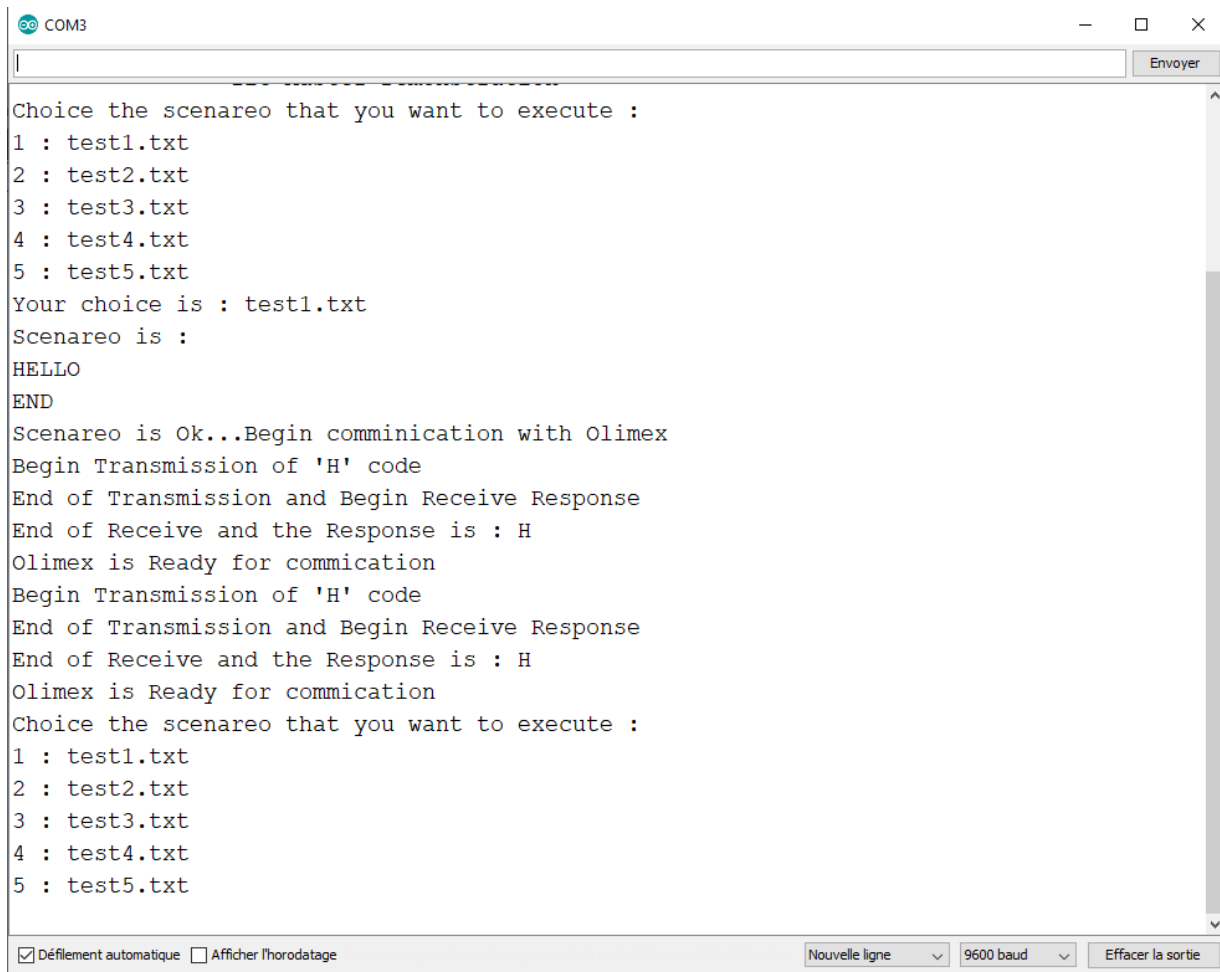
// A STOP condition or repeated START condition has been received while still addressed as Slave : 0xA0
else if ((TWSR & 0xF8) == TW_STOP)
{
    #if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0
        for (unsigned char i = 0; i <= strlen(MSG15); i++)
        {
            UART_Transmit(MSG15[i]);
        }
    #endif
    TWCR |= (1<<TWEA) | (1<<TWEN);
    //TWCR |= (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN);
}

// if the code transmitted by the Master is not equal to one of the precedent condition
else
{
    #if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0
        for (unsigned char i = 0; i <= strlen(MSG16); i++)
        {
            UART_Transmit(MSG16[i]);
        }
    #endif
    TWCR |= (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN);
}
}
```

Annexe 12 : Programme du microcontrôleur ATmega128A

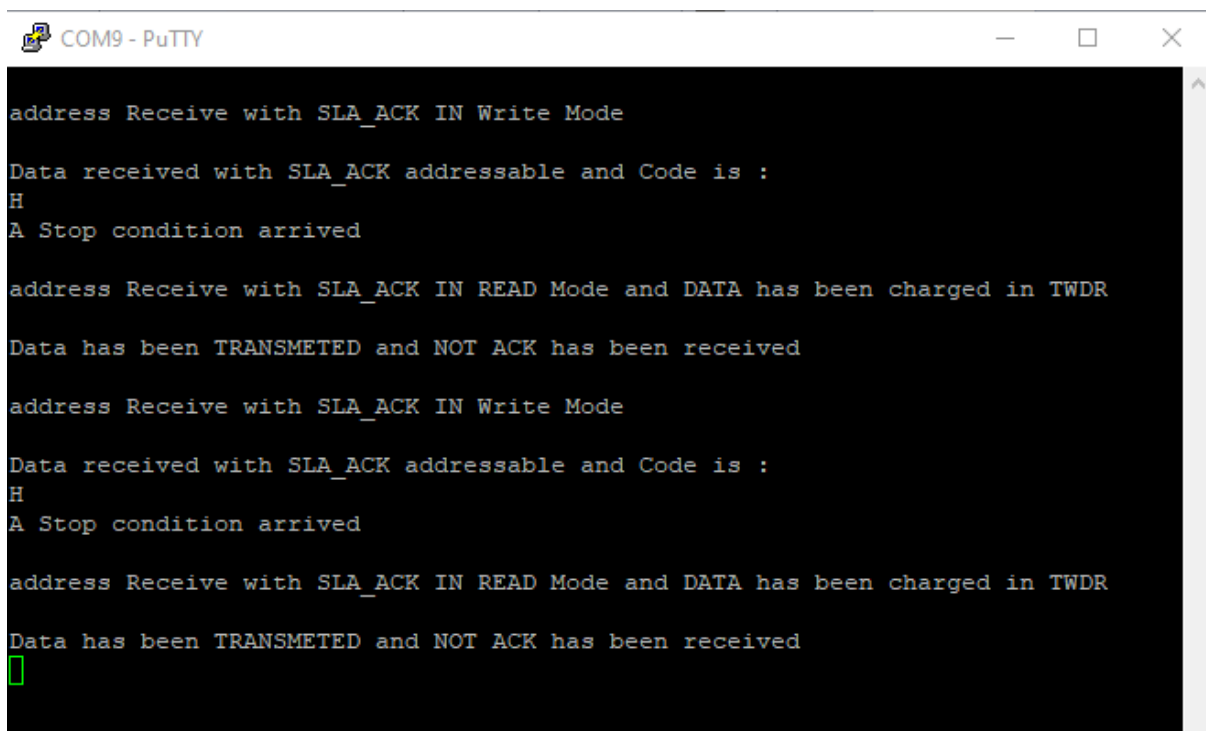
```
int main(void)
{
    PORT_Init(); // Initialisation des ports
    #if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0 // If debug defined then initiate UART
        UART_Init(9600);
    #endif
    TWI_SLAVE_INIT(); // Initialization for i2c
    sei(); // Enable general interrupt bit
    #if defined(AVR_DEBUG_TWI) && AVR_DEBUG_TWI > 0
        for (unsigned char i = 0; i <= strlen(MSG17) ; i++)
        {
            UART_Transmit(MSG17[i]);
        }
    #endif
    while (1)
    {
        if (CODE == 'M')
        {
            PrevC = 'M';
            STATE = 'M';
            CHPT = 0;
            //UART_Transmit(STATE);
            for (j = 0 ; j < 40 ; j++) // generate the measure
            {
                UART_Transmit('A');
                CHPT += 1;
                DATA[j] = CHPT;
            }
            j = 0;
            CODE = 'V'; // V for Vide
            STATE = 'R';
            _delay_ms(5);
        }
    }
}
```

Arduino



```
Choice the scenareo that you want to execute :
1 : test1.txt
2 : test2.txt
3 : test3.txt
4 : test4.txt
5 : test5.txt
Your choice is : test1.txt
Scenareo is :
HELLO
END
Scenareo is Ok...Begin comminication with Olimex
Begin Transmission of 'H' code
End of Transmission and Begin Receive Response
End of Receive and the Response is : H
Olimex is Ready for commication
Begin Transmission of 'H' code
End of Transmission and Begin Receive Response
End of Receive and the Response is : H
Olimex is Ready for commication
Choice the scenareo that you want to execute :
1 : test1.txt
2 : test2.txt
3 : test3.txt
4 : test4.txt
5 : test5.txt
```

Olimex



```
address Receive with SLA_ACK IN Write Mode
Data received with SLA_ACK addressable and Code is :
H
A Stop condition arrived
address Receive with SLA_ACK IN READ Mode and DATA has been charged in TWDR
Data has been TRANSMETED and NOT ACK has been received
address Receive with SLA_ACK IN Write Mode
Data received with SLA_ACK addressable and Code is :
H
A Stop condition arrived
address Receive with SLA_ACK IN READ Mode and DATA has been charged in TWDR
Data has been TRANSMETED and NOT ACK has been received
█
```

Arduino

```
Choice the scenareo that you want to execute :
1 : test1.txt
2 : test2.txt
3 : test3.txt
4 : test4.txt
5 : test5.txt
Your choice is : test2.txt
Scenareo is :
HELLO
GETMEASURE
END
Scenareo is Ok...Begin comminication with Olimex
Begin Transmission of 'H' code
End of Transmission and Begin Receive Response
End of Receive and the Response is : H
Olimex is Ready for commication
Didn't ask for Measure..Scenareo is not valid
Didn't ask for Measure..Scenareo is not valid
Choice the scenareo that you want to execute :
```

Olimex

```
address Receive with SLA_ACK IN Write Mode

Data received with SLA_ACK addressable and Code is :
H
A Stop condition arrived

address Receive with SLA_ACK IN READ Mode and DATA has been charged in TWDR

Data has been TRANSMETED and NOT ACK has been received
```



Arduino

```
Choice the scenareo that you want to execute :
1 : test1.txt
2 : test2.txt
3 : test3.txt
4 : test4.txt
5 : test5.txt
Your choice is : test3.txt
Scenareo is :
HELLO
STARTMESURE
GETSTATE
END
Scenareo is Ok...Begin comminication with Olimex
Begin Transmission of 'H' code
End of Transmission and Begin Receive Response
End of Receive and the Response is : H
Olimex is Ready for commication
Begin Transmission of 'M' code
Transmission of 'M' code is done
Begin Transmission of 'S' code
Receive Response :
Measure is Ready
```

Olimex

```
address Receive with SLA_ACK IN Write Mode

Data received with SLA_ACK addressable and Code is :
M
A Stop condition arrived
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
address Receive with SLA_ACK IN Write Mode

Data received with SLA_ACK addressable and Code is :
S
A Stop condition arrived

address Receive with SLA_ACK IN READ Mode and DATA has been charged in TWDR

Data has been TRANSMETED and NOT ACK has been received

address Receive with SLA_ACK IN Write Mode

Data received with SLA_ACK addressable and Code is :
S
A Stop condition arrived

address Receive with SLA_ACK IN READ Mode and DATA has been charged in TWDR

Data has been TRANSMETED and NOT ACK has been received
```

Arduino

```
Your choice is : test4.txt
Scenareo is :
HELLO
STARTMESURE
GETSTATE
GETMESURE
END
Scenareo is Ok...Begin comminication with Olimex
Begin Transmission of 'H' code
End of Transmission and Begin Receive Response
End of Receive and the Response is : H
Olimex is Ready for commication
Begin Transmission of 'M' code
Transmission of 'M' code is done
Begin Transmission of 'S' code
Receive Response :
Measure is Ready
Begin Transmission of 'G' code
End of Transmission and Begin Receiving data
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17


---


18
19
20
All Measure request has been received


---


```

Olimex

Annexe 13 : Simulation des cinq scénarios – scénario4

```

address Receive with SLA_ACK IN Write Mode

Data received with SLA_ACK addressable and Code is :
H
A Stop condition arrived

address Receive with SLA_ACK IN READ Mode and DATA has been charged in TWDR

Data has been TRANSMETED and NOT ACK has been received

address Receive with SLA_ACK IN Write Mode

Data received with SLA_ACK addressable and Code is :
M
A Stop condition arrived
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
address Receive with SLA_ACK IN Write Mode

Data received with SLA_ACK addressable and Code is :
S
A Stop condition arrived

address Receive with SLA_ACK IN READ Mode and DATA has been charged in TWDR

Data has been TRANSMETED and NOT ACK has been received

address Receive with SLA_ACK IN Write Mode

Data received with SLA_ACK addressable and Code is :
G
A Stop condition arrived

address Receive with SLA_ACK IN READ Mode and DATA has been charged in TWDR

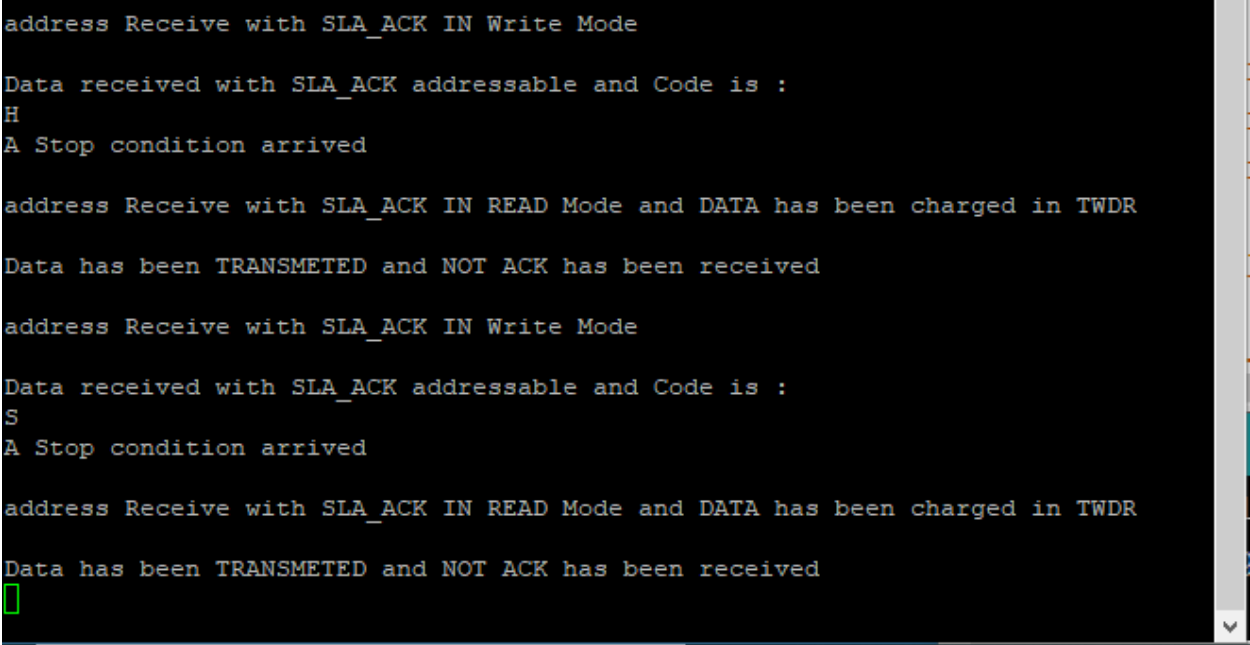
```

[illegible]

Arduino

```
Your choice is : test5.txt
Scenareo is :
HELLO
GETSTATE
GETMESURE
END
Scenareo is Ok...Begin communication with Olimex
Begin Transmission of 'H' code
End of Transmission and Begin Receive Response
End of Receive and the Response is : H
Olimex is Ready for communication
Begin Transmission of 'S' code
Receive Response :
Error message has been received..Scenareo is not valid..may ask Olimex to start mesure
```

Olimex



```
address Receive with SLA_ACK IN Write Mode

Data received with SLA_ACK addressable and Code is :
H
A Stop condition arrived

address Receive with SLA_ACK IN READ Mode and DATA has been charged in TWDR

Data has been TRANSMETED and NOT ACK has been received

address Receive with SLA_ACK IN Write Mode

Data received with SLA_ACK addressable and Code is :
S
A Stop condition arrived

address Receive with SLA_ACK IN READ Mode and DATA has been charged in TWDR

Data has been TRANSMETED and NOT ACK has been received
█
```