TP

Communication série et Bus de terrain

IGUERNELALA Lila
BELKADA Karima

15/10/2023

—

ESIEA

Table des matières

I. Introduction

Lors du TP et durant cette semaine, on a acquis des connaissances sur les concepts fondamentaux de la communication série, ainsi que les principes de transmission des données sur un canal bidirectionnel avec des signaux binaires.

On a pu apprendre et se familiariser avec l'utilisation du bus RS485, qui est très utilisé pour la communication industrielle à longue distance.

On a appris à définir un format de trames, on a également développé des compétences dans la planification de projets, y compris la construction d'une messagerie.

II. Principe de fonctionnement de la RS485

- La RS485 permet la communication série fiable entre plusieurs équipements sur un bus partagé. Elle est généralement utilisée dans les applications industrielles car elle est robuste, fiable et pas cher et elle a la capacité de prendre en charge les longues distances.
  Elle implique généralement un équipement maître qui contrôle la communication avec des équipements esclaves. Les trames de données suivent un format standard avec une gestion des erreurs pour garantir la fiabilité de la communication.
  Le FT231X est utilisé pour convertir des données USB en signaux UART (série) qui sont ensuite transmis sur la ligne RS485.

- Après avoir terminé d'analyser la carte RS485, on a branché les cartes par câble USB à l'ordinateur.
  Vous utiliserez le terminal PuTTY pour envoyer et recevoir des caractères. Cela simule la communication entre deux dispositifs via RS485.
  Par la suite on a utilisé un oscilloscope pour observer les signaux électriques réels transmis sur la ligne RS485 lorsqu'on envoie des caractères majuscules "U".

III. Développement d'un protocole de communication

On a mis à jour la classe "CSerial" en implémentant des fonctions spécifiques pour effectuer ces actions, et ceci, conformément aux directives du guide de programmation "D2XX". Ces méthodes ajouteront de nouvelles fonctionnalités à notre projet de communication, ça va nous permettre de configurer et de contrôler la communication série de manière plus précise et de transmettre et de recevoir des données de manière plus efficace.

Les figures (1,2,3,4) illustrent les programmes .h et .cpp

```
1   //Serial.h
2
3   #pragma once
4   #include "ftd2xx.h"
5   class CSerial
6   {
7   public:
8     char numSerie[20]; // Le numéro de série de la carte
9     char txBuffer[255];
10    char rxBuffer[255];
11    //Constructor
12    CSerial(void);
13    //Destructor
14    ~CSerial();
15    //Print all FTDI devices serial number
16    void listDevice(void);
17
18    //Open communication
19    bool open(char * p_serialNumber);
20
21    //Close communication
22    void close(void);
23
24    // Retourne le nb de cartes connectées. -1 si erreur
25    DWORD CSerial::ret_nb_cartes_connectees(void);
26
27    // Retourne le numéro de série de la carte connectée : 1, 2, ..
28    // Curieusement cette méthode ne peut être appelée qu'une seule fois
29    // Retourne -1 si erreur, 0 sinon
30    char* CSerial::ret_num_serie(DWORD numCarte);
31    void  CSerial::configurer(void);
32    bool  CSerial::Write(DWORD nbCar);
33    void CSerial::read(DWORD dwBytesToRead);
34
35  protected:
36     FT_STATUS m_ftStatus;
37     FT_HANDLE m_ftHandle;
38     DWORD baudRate;
39  public:
40      void setBaudRate(DWORD baudRate);
41  };
42
```

Figure 1 : programme « Serial.h »

```cpp
//Serial.cpp
#include "stdafx.h"

#include "serial.h"

//Constructor
CSerial::CSerial(void)
{
 m_ftStatus=FT_OK;
 m_ftHandle=NULL;
}

//Destructor
CSerial::~CSerial()
{
 close();
}
//Initilisation
void CSerial::configurer(void)
{
    m_ftStatus = FT_SetBaudRate(m_ftHandle, 9600);
    if (m_ftStatus == FT_OK)
    {
        printf("OK !");
    }
    else {
        printf("lol");

    }
    printf("\n");

    m_ftStatus = FT_SetDataCharacteristics(m_ftHandle, FT_BITS_8, FT_STOP_BITS_1, FT_PARITY_NONE);
        if (m_ftStatus == FT_OK)
        {
            printf("OK!");
        }
        else {
            printf("lol");
        }
    printf("\n");
}

//Print all FTDI devices serial number
void CSerial::listDevice(void)
{
 FT_STATUS l_ftStatus;
 DWORD l_numDevs;

 l_ftStatus = FT_ListDevices(&l_numDevs,NULL,FT_LIST_NUMBER_ONLY);
 if (l_ftStatus == FT_OK)
 {
  char l_buffer[64];
  for (DWORD l_devIndex=0;l_devIndex<l_numDevs;l_devIndex++)
  {
   l_ftStatus = FT_ListDevices((PVOID)l_devIndex,l_buffer,FT_LIST_BY_INDEX|FT_OPEN_BY_SERIAL_NUMBER);
   if (l_ftStatus == FT_OK)
   {
    printf("%d: '%s'\n",l_devIndex,l_buffer);
   }
  }
 }
}
```

```
67      //Open communication
68      /// Return true if successful
69    bool CSerial::open(char * p_serialNumber)
70    {
71      m_ftStatus = FT_OpenEx(p_serialNumber,FT_OPEN_BY_SERIAL_NUMBER,&m_ftHandle);
72      if  (m_ftStatus==FT_OK)
73        return true;
74      return false;
75    }
76      //Read Function
77    void CSerial::read(DWORD dwBytesToRead)
78    {
79            DWORD EventDWord;
80            DWORD TxBytes;
81            DWORD BytesReceived;
82            m_ftStatus = FT_Read(m_ftHandle, &TxBytes, dwBytesToRead, &BytesReceived);
83                if (m_ftStatus != FT_OK) {
84                    printf("Failed");
85                }
86
87            }
88
89
90      //Write Function
91    bool CSerial::Write(DWORD nbCar)
92    {
93       DWORD BytesWritten = 0;
94            m_ftStatus = FT_Write(m_ftHandle, txBuffer, nbCar,  &BytesWritten);
95            if (m_ftStatus != FT_OK) {
96                printf("Failed");
97            }
98            return true;
99
100   }
101     //Close communication
102   void CSerial::close(void)
103   {
104     if (m_ftHandle!=NULL)
105       FT_Close(m_ftHandle);
106     m_ftHandle=NULL;
107   }
108
```

```
108
109   DWORD CSerial::ret_nb_cartes_connectees(void)
110   {
111       DWORD numDevs = 0;
112       FT_STATUS ftStatus = FT_ListDevices(&numDevs, NULL, FT_LIST_NUMBER_ONLY);
113       if (ftStatus == FT_OK) {
114           // FT_ListDevices OK, number of devices connected is in numDevs
115       }
116       else {  // FT_ListDevices failed
117           return -1;
118       }
119
120       return numDevs;
121   }
122
123   char* CSerial::ret_num_serie(DWORD numCarte)
124   {
125       DWORD devIndex = numCarte - 1; // first device char Buffer[64]; // more than enough room!
126       FT_STATUS ftStatus = FT_ListDevices((PVOID)devIndex, numSerie, FT_LIST_BY_INDEX | FT_OPEN_BY_SERIAL_NUMBER);
127       if (ftStatus == FT_OK) {
128           // FT_ListDevices OK, serial number is in Buffer
129           return numSerie;
130       }
131       else {
132           // FT_ListDevices failed
133           printf("!!! ret_num_serie() : Impossible enumerer les cartes\n");
134           return numSerie;
135       }
136
137   }
138
```

Figure 2 : programme « Serial.cpp »

- On a complété le code « communication.cpp » qui est le point de départ de l'exécution de notre programme. C'est là où notre programme commence son exécution et où le code initial est exécuté.

```
1   // Communication.cpp : Defines the entry point for the console application.
2   //
3
4   #include "stdafx.h"
5   #include "serial.h"
6
7   void transmission(char letter, CSerial carte_1, CSerial carte_2)
8   {
9       for (int i = 0; i < 26; i++)
10      {
11          carte_1.txBuffer[0] = letter + i;
12
13          printf("Round %d : ", i);
14          if ((i % 2) == 0) {
15              carte_1.Write(1);
16              carte_1.read(1);
17              carte_2.read(1);
18              carte_2.Write(1);
19              carte_2.txBuffer[0] = carte_1.txBuffer[0];
20
21              printf("C1=%c  ", carte_1.txBuffer[0]);
22              printf("C2=%c  ", carte_2.txBuffer[0]);
23              carte_2.txBuffer[0] = carte_1.txBuffer[0] + 1;
24          }
25          else {
26              carte_2.Write(1);
27              carte_2.read(1);
28              carte_1.read(1);
29              carte_1.Write(1);
30              carte_1.txBuffer[0] = carte_2.txBuffer[0];
31
32              printf("C1=%c  ", carte_1.txBuffer[0]);
33              printf("C2=%c\n", carte_2.txBuffer[0]);
34          }
35      }
36  }
37
```

Figure 3 : programme « communication.cpp »

```
38
39  int _tmain(int argc, _TCHAR* argv[])
40  {
41      CSerial carte_1;
42      CSerial carte_2;
43      DWORD nbCartesConnectees = 0;
44
45      //Print all devices serial numbers
46      carte_1.listDevice();
47      //carte_2.listDevice();
48
49      nbCartesConnectees = carte_1.ret_nb_cartes_connectees();
50      printf("Nombre de cartes connectees : %d\n", nbCartesConnectees);
51
52      // Ouverture de la carte
53      // Soit vous indiquez en argument le numéro de série de la carte, exemple : l_serial.open("DAYLHURK")
54      //
55      // Soit vous récupérez le numéro de série dynamiquement.
56      // En argument, 1 pour la première carte, 2 pour la 2e
57      // C'est plus pratique si vous changez de carte entre les séances de TP
58      if (carte_1.open(carte_1.ret_num_serie(1))==true && carte_2.open(carte_2.ret_num_serie(2)) == true)
59      {
60          printf("Carte 1 ouverte avec succes\n");
61          printf("Carte 2 ouverte avec succes\n");
62              printf("La carte 1 ouverte a pour numero de serie : %s\n", carte_1.numSerie);
63
64
65              printf("La carte 2 ouverte a pour numero de serie : %s\n", carte_2.numSerie);
66
67              carte_1.configurer();
68              carte_2.configurer();
69              char kb = _getch();
70              while (kb != 'q' && kb != 'Q')
71              {
72                  transmission('a', carte_1, carte_2);
73                  kb = _getch();
74              }
75              if (kb == 'q' || kb == 'Q') { printf("DONE");}
76      }
77      else {
78          printf("!!! Echec ouverture de la carte 1\n");
79          printf("!!! Echec ouverture de la carte 2\n");
80      }
81          // if (carte_2.open(carte_2.ret_num_serie(2)) == true)
82          //{
83              //printf("la carte 2 ouverte avec succes\n");
84                  //printf("la carte 2 ouverte a pour numero de serie : %s\n", carte_2.numSerie);
85
86  // }
87          // else {
88              // printf("!!! Echec ouverture de la carte 2\n");
89          //}
90
91      //initialiser la carte
92      //carte_1.configurer();
93      //carte_1.txBuffer[0] = 'a';
94      //carte_1.Write(1);
95      //carte_1.read(1);
96
97  // carte_2.configurer();
98  // carte_2.txBuffer[0] = 'b';
99      //carte_2.Write(1);
100     //carte_2.read(1);
101
102     //close comm.
103     carte_1.close();
104     carte_2.close();
105     _getch();
106       return 0;
107 }
108
```

Figure 4 : Programme « communication.cpp » (suite)

IV. Le résultat de simulation du Ping Pong :

On a complété les deux programmes (voir les figures 1,2,3,4,5) afin d'assurer la communication entre eux, en envoyant et en recevant des caractères via la RS485.

Lorsqu'on appuie sur une touche ("a" par exemple), le premier programme envoie la lettre au second programme qui l'affiche à l'écran et renvoie le code de caractère reçu augmenté, le second programme renvoie donc cette nouvelle lettre au premier programme,

Le premier programme réagit de la même manière, il reçoit la nouvelle lettre, l'affiche et l'incrémente à une autre lettre puis la renvoie.

Ce cycle d'incrémentation de lettres se répète jusqu'à ce qu'on atteigne la lettre "z". Et si jamais on appuie à nouveau sur la lettre "a", le cycle recommence avec la lettre "a" (voir figure 6)

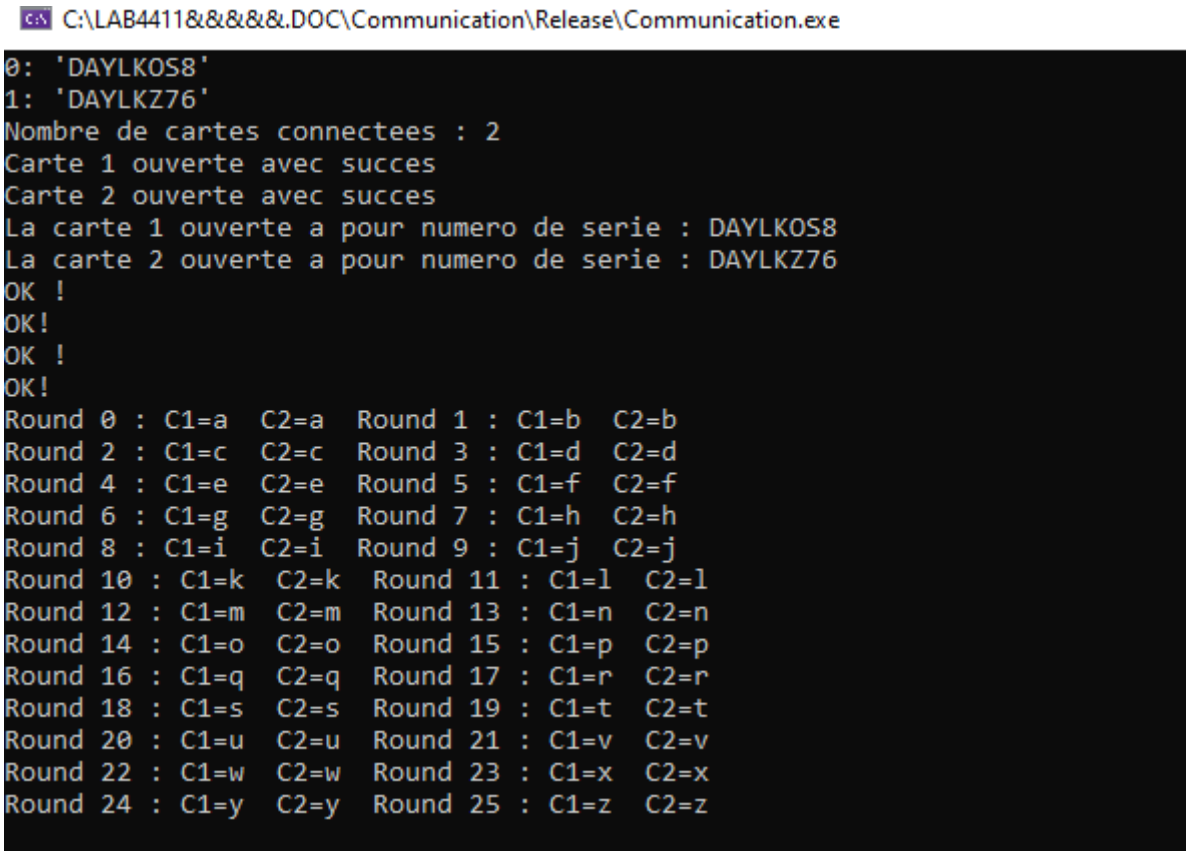

Figure 5 : résultats de simulation

Le cycle de communication continue tant que la lettre 'z' n'est pas atteinte, et il est réinitialisé lorsqu'on appuie sur une touche pour le relancer.

Pour arrêter les deux programmes, on a prévu la touche "q", lorsqu'elle est pressée, elle provoque la fermeture des deux programmes.

V. Communication multipoints RS485

La communication dans ce cas est bidirectionnelle, ceci veut dire que le maître peut envoyer des commandes aux esclaves, et les esclaves peuvent à leur tour envoyer des données en retour au maître. En cas de perte de trame, ça va se répéter automatiquement afin de garantir que le message est bien transmis avec succès.

Pour assurer une communication fiable entre plusieurs systèmes connectés via un bus a deux fils de type RS485, nous avons choisi d'utiliser le protocole de communication série MODBUS RTU.

A. Principe de fonctionnement :

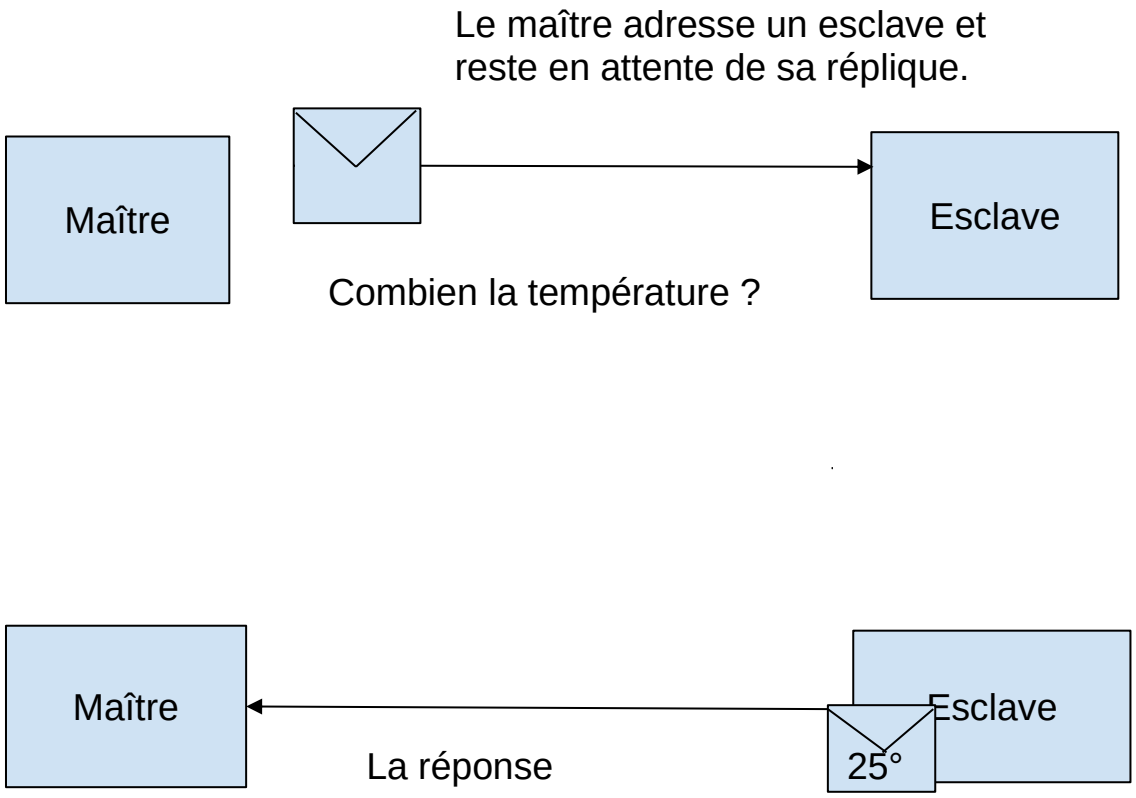

Figure 6 : principe du fonctionnement du MODBUS RTU

B. MODBUS RTU a deux types de transmission :

1. Mode HALF_DUPLEX :

Dans ce type, le maître peut envoyer et recevoir seulement en deux fils avec deux conducteur “Rx=>Réception” et “Tx=>Émission”, cette transmission est instantanée ou les données sont communiquées l’une après l’autre

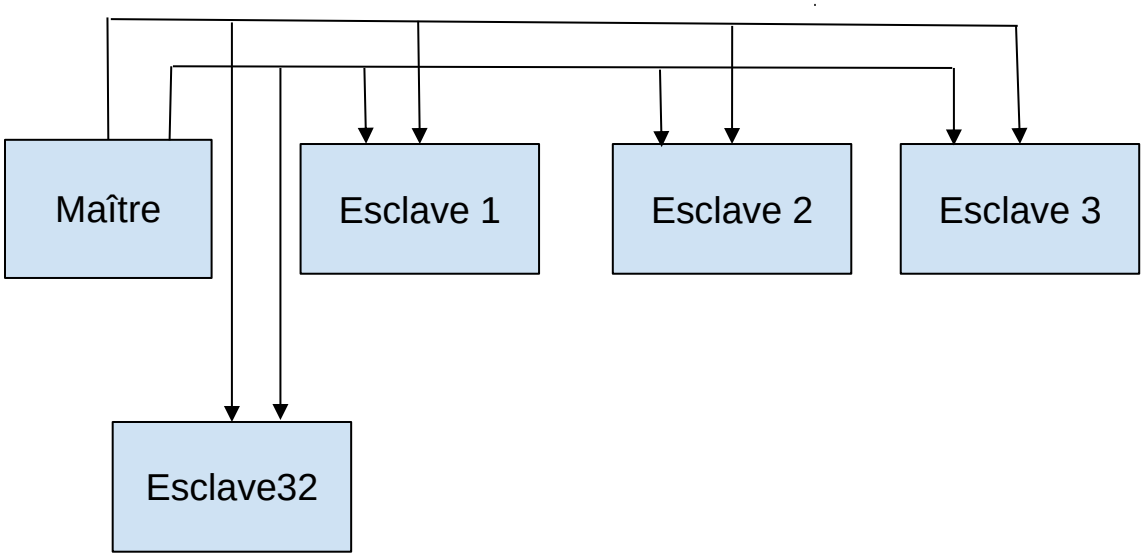

Figure 7 : La transmission HALF_DUPLEX

2. Mode FULL_DUPLEX :

Les dispositifs peuvent envoyer des données simultanément sur le même canal de communication. C'est également utilisé dans des communication série lorsque des équipements sont configurés pour une communication bidirectionnelle simultanée, offrant ainsi une communication rapide et ininterrompue dans les deux sens.

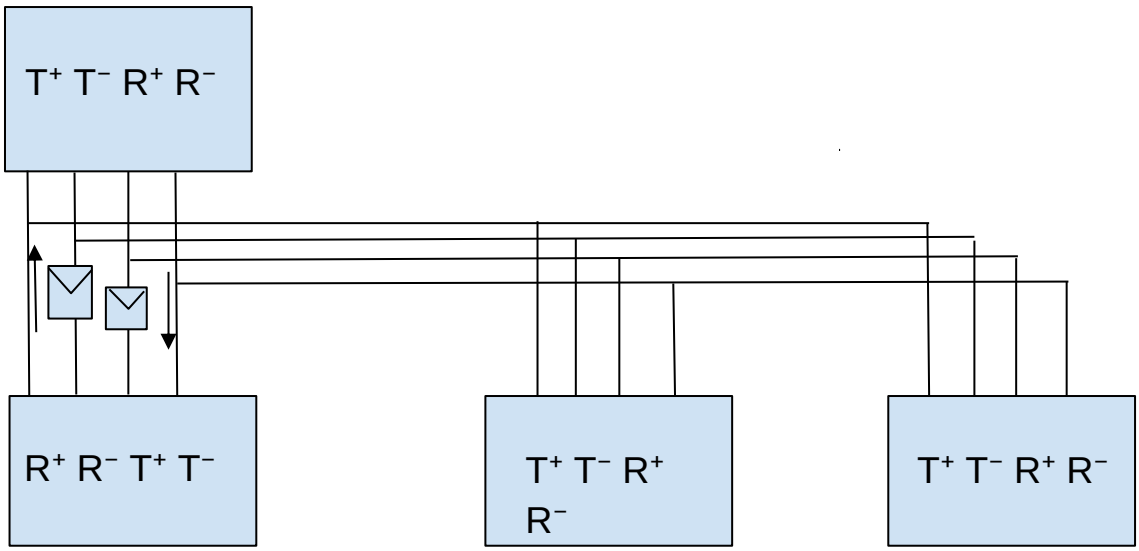

Figure 8 : La transmission FULL_DUPLEX

- Nous opterons pour l'utilisation de trames MODBUS RTU comme point de départ de notre communication entre le maître et les esclaves. Ces trames seront configurées pour contenir des informations essentielles, telles que l'adresse de l'esclave cible, le code de fonction, les données, un champ de vérification, ainsi que les bits de start, de stop et de parité. Cela servira de fondement pour notre communication.

Le format d'une trame :

Silence	Adresse Esclave	Code Requête	Données	Contrôle	Silence
Start	Adresse	Code Fonction	Data	CRC	Stop
1 Bit	1 Octet	1 Octet	256 Octets	2 Octets	1 Bit

Figure 9 : format d'une trame MODBUS RTU

Explication de la structure :

Bit de Start : Le bit de Start cherche une adresse d'un esclave (caméra, automate…) pour envoyer une requête(message)

Adresse : L 'adresse cible permet de diriger l'information vers son destinataire spécifique.

Code Fonction : Le code de fonction permet la lecture et l'écriture d'informations que ce soit à partir des capteurs ou à partir d'autres équipements tels que des compteurs électriques, de stations météorologiques, de dispositifs de contrôle de température, et d'autres équipements de surveillance.

Data : Les données sont encapsulées dans la Trame en vue de leurs transmissions

CRC : Le CRC est un mécanisme de contrôle de données entre l'émetteur et le récepteur via la parité.

Stop : Le Stop dans la trame est utilisé pour marquer la fin de la transmission de données.

Exemple d'un MODBUS RTU :

Dans cet exemple, le maître envoie une requête au dispositif esclave (adresse 0x2) pour obtenir des données de température. Le code de fonction 0x04 est utilisé pour demander la lecture de la température. L'esclave traite la demande, puis répond avec les données de température (0x0A, 0x0B). Le CRC est calculé pour vérifier l'intégrité de la trame.

Etape	Maître (M)	Esclave(E)
1	**Envoie :** Bit de Start : 0x3 Adresse : 0x2 Code de fonction :0x4 Données : 0x00, 0x1, CRC : calculé Bit de Stop : 0x4	**Réception :** Bit de Start : 0x3 Adresse : 0x2 Code de fonction :0x04 Données : 0x0A, 0x0B, CRC : **calculé** Bit de Stop : 0x4 Traitement de la demande
2	Réponse : Bit de Start : 0x3 Adresse : 0x2	Réception : Bit de Start : 0x3 Adresse : 0x2
3	Code de Fonction : 0x4 Données : 0x0A, 0x0B CRC : **calculé** Bit de Stop : 0x4	Code de Fonction : 0x4 Données : 0x00, 0x01 CRC : **calculé**

C. MODBUS TCP/IP :

1. Mode de fonctionnement :

Modbus TCP/IP est un protocole de communication largement utilisé dans les systèmes d'automatisation industrielle. Il fonctionne en établissant une connexion TCP/IP entre un client (généralement un superviseur ou un contrôleur) et un serveur (l'esclave Modbus).

Seuls les clients sont actifs, le serveur est complètement passif, ce sont les clients qui doivent lire et écrire. Chaque client doit se connecter au serveur en protocole TCP (adresse IP dure d'un serveur).

Remarque : Le Maître est le client, l'esclave est le serveur.

Le schéma ci-dessous illustre le fonctionnement de Modbus TCP/IP

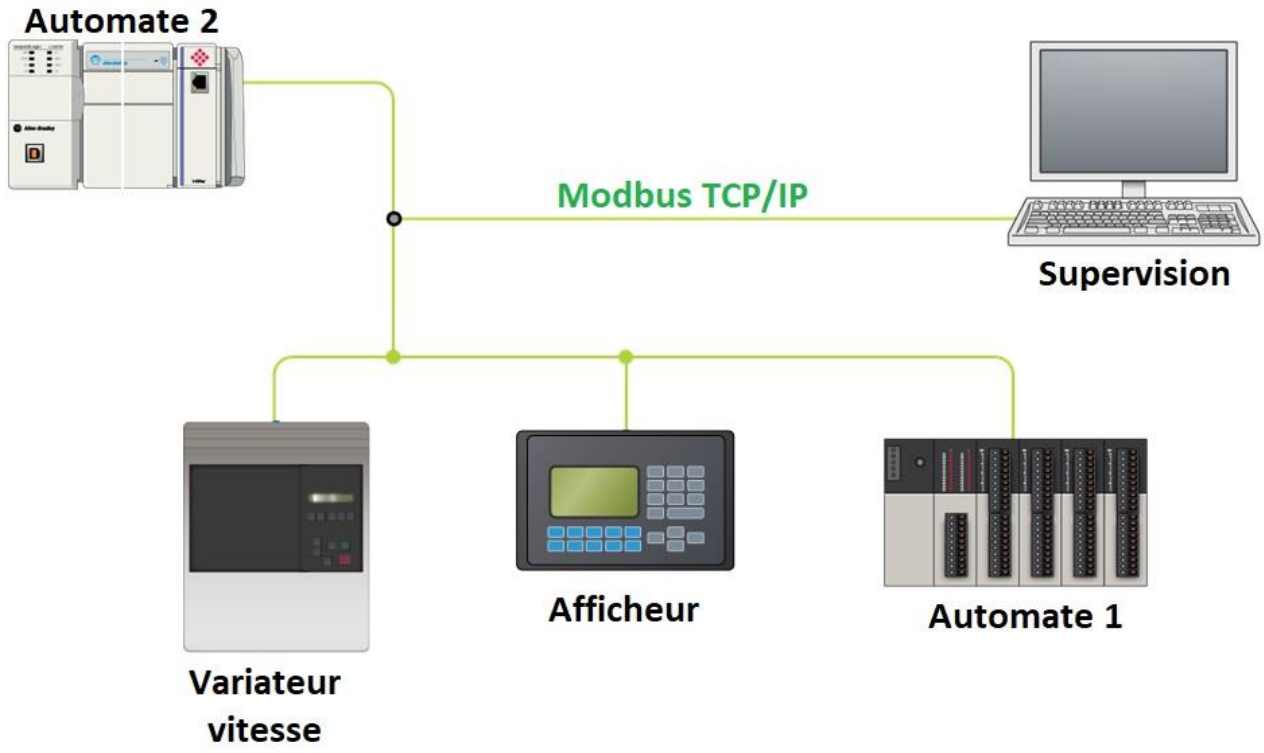

Figure 10 : le fonctionnement de Modbus TCP/IP

Exemple :

Dans cet exemple de communication Modbus, il est possible d'émettre une requête de lecture de température en mode broadcast à l'ensemble des esclaves en utilisant l'adresse d'esclave 0. Dans ce mode de diffusion, tous les esclaves ont la possibilité de répondre, mais il n'existe aucune garantie quant à la réception de l'ensemble des réponses

Etape	Client (Maître)	Serveur (Esclave)
1	**Envoie une requête de lecture de température en broadcast (Adresse 0)**	
2		**Reçoit la requête et traite la demande.**
3		**Réponse : (Temp : 25℃)**

Remarque :

Le client envoie une requête de lecture de température en mode broadcast à tous les esclaves (en utilisant l'adresse d'esclave 0). Le serveur qui reçoit la demande traite la requête et renvoie une réponse avec la température 25°C, cependant, tous les esclaves peuvent répondre à la requête, mais il n'y a aucune garantie que toutes les réponses seront reçues en raison du mode de diffusion.

Résumé :

MODBUS RTU est principalement utilisé pour la communication série dans des environnements industriels, tandis que MODBUS TCP/IP est conçu pour fonctionner sur des réseaux Ethernet. Tous les deux sont largement utilisés dans l'automatisation industrielle, mais ils diffèrent dans leurs méthodes de communication, leurs vitesses et leurs niveaux de sécurité. Le choix entre les deux dépend des besoins spécifiques d'une application.