TP Système : Les Cloches

Sommaire

1. Introduction :

Nous avions pour projet de concevoir un programme qui puisse communiquer avec une carte réseau ETZ-410. Le but était d'envoyer une trame à cette automate pour faire sonner les différentes cloches.

2. Questions préliminaires :

1) Pour communiquer avec l'automate, nous devions lui envoyer une trame modbus encapsulée dans une trame TCP/IP. Le protocole modbus se trouve dans la couche application du modèle OSI.

- Encapsulation :

Le protocole modbus peut être encapsulé par les supports suivants :

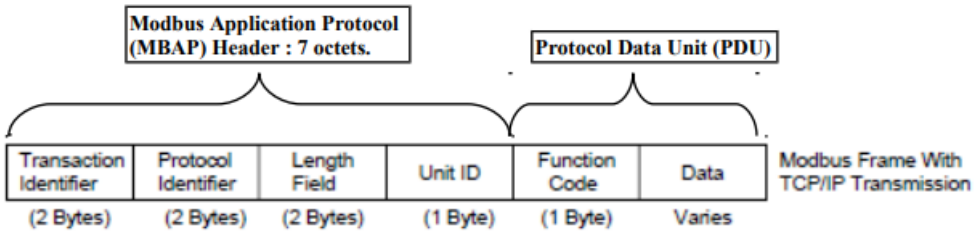
- RS-232
- RS-485
- RS-422
- Ethernet TCP/IP (via un câble ethernet, c'est ce que nous avons choisi)

- Mode de fonctionnement du protocole modbus TCP :

Le maître parle à un esclave et attend sa réponse ou le maître parle à l'ensemble des esclaves, sans attendre de réponse (diffusion générale). Il ne peut y avoir sur la ligne qu'un seul équipement en train d'émettre. Aucun esclave ne peut envoyer un message sans une demande préalable du maître. Le dialogue entre les esclaves est impossible.

Nous avons utilisé un mode **half-duplex** c'est-à-dire qu'il n'y a qu'une seule communication à la fois (soit émission soit réception).

Ci_dessous, un schéma représentant une trame modbus



La requête contient :

1- L'adresse de l'esclave à interroger (le serveur)
2- Le code fonction qui indique le type d'action à exécuter (la lecture des bits)
3- La plage de bits concernée (elle dépend du code fonction vu précédemment)
4- Les données à écrire si on doit écrire

La réponse contient :

1- L'adresse de l'esclave qui répond
2- Un code fonction qui indique l'action effectuée
3- Le nombre d'octets de données dans la réponse
4- Les données lues (si il y a eu une lecture)

2) Le principe du protocole TCP/IP client serveur repose sur une communication entre deux machines, le client et le serveur.

   Le client initie l'échange et envoie une requête
   Les serveur est en permanence en écoute sur son port, dès qu'il détecte une requête, il la traite et répond au client.

   Un serveur doit être capable de réponde à plusieurs clients en même temps.

   ETZ-510 est donc un serveur car elle ne fait que répondre à nos requêtes.

```
char trame[14];

trame[0] = 0x00;
trame[1] = 0x00;
trame[2] = 0x00;
trame[3] = 0x00;
trame[4] = 0x00;
trame[5] = 0x06;
trame[6] = 0x00;
trame[7] = 0x06;
trame[8] = 0x00;
trame[9] = 0x02;
trame[10] = 0x00;
trame[11] = 0x12;
trame[12] = 0x01;
trame[13] = 0x01;

liaison->sendsocket(trame);
```

3)

La trame suivante permet de faire sonner le cloche 1.

Partie 2 – Besoin et Contraintes du Projet

1. Démarrage du TP :

Nous avons tout d'abord du trouver un moyen de communiquer avec la carte réseau.

Pour cela nous nous sommes connecté en Ethernet à la carte via une de nos machines.

Nous avons modifié la configuration réseau de notre machine en nous attribuant une IP fixe pour être sur la même plage d'IP que la carte. Nous avons également mis le même masque que l'IP de la carte.

Mr Langlace nous a aidé à rebrasser une prise réseau orange, car la l'automate était sur le réseau orange.

Nous avons tout d'abord testé la connexion entre notre machine et la carte via Hercule.

Pour réaliser ce TP, nous avons crée une classe Liaisontcp Pour organiser notre code et implémenter des méthodes paramétrant la connexion.

```
#ifndef LiaisontcpH
#define LiaisontcpH

class Liaisontcp
{
    private:

              int socket_to_send;
              char* ip;
              int port;

   public:
              Liaisontcp(char* ip1,int port1);
              int createsocket();
              bool sendsocket(char* buffer); //en mode connect ;)
              bool receivsocket(char* buffer,int taille,int flag); //en mode connect
              bool connect_socket();
              void closeSocket();




};
#endif
```

L'objet Liaisontcp se construit à l'aide des variables suivantes :

- Un entier ''socket_to_send'' qui sera notre socket utilisé pour la connexion à l'automate et pour l'envoi de données

- Un tableau de caractère ''ip'' qui stockera l'ip du maître (l'ip de notre machine)

- Un entier ''port'' qui stockera le port du serveur (esclave) qui sera 502

Createsocket() :

```
#include "Liaisontcp.h"
#include <winsock2.h>
#include <stdio.h>
#pragma comment(lib,"ws2_32.lib")
#pragma package(smart_init)

int Liaisontcp::createsocket()
{
    WSADATA wsaData;

    WSAStartup(MAKEWORD(2,2), &wsaData);

    int sock = socket(AF_INET,SOCK_STREAM,0);

    if(sock==INVALID_SOCKET)
    {

       return -1;
    }
    else
    {
       socket_to_send = sock;
       return sock;
    }
}
```

C'est à partir de la fonction **socket** que nous créons notre variable.

SOCK_STREAM : paramètre utilisé pour un type de socket qui fournit des flux d'octets séquencés. Ce type de socket utilise le protocole TCP (c'est pour cela qu'on l'a choisi)

WSDATA : structure qui contient les informations de l'implémentation d'un socket windows.

Cette méthode nous retourne un socket crée (type SOCKET = type int) ou nous renvoie -1 en cas d'erreur.

Connect_socket() :

Dans cette méthode nous configurons les structures hostent et SOCKADDR_IN pour établir la connexion.

Si la connexion échoue on retourne faux, sinon on retourne vrai

```
bool Trame::connect_socket()
{

    struct hostent *hostinfo = NULL;
    SOCKADDR_IN sin = { 0 }; /* initialise la structure avec des 0 */
    const char *hostname = "192.168.1.80";

    hostinfo = gethostbyname(hostname);

    if (hostinfo == NULL) /* l'hôte n'existe pas */
    {
        fprintf (stderr, "Unknown host %s.\n", hostname);
        exit(EXIT_FAILURE);

    }

    sin.sin_addr = *(IN_ADDR *) hostinfo->h_addr; /* l'adresse se trouve dans le champ h_addr de la structure hostinfo */
    sin.sin_port = htons(502); /* on utilise htons pour le port */
    sin.sin_family = AF_INET;

    if(connect(socket_to_send,(SOCKADDR*) &sin,sizeof(SOCKADDR)) == SOCKET_ERROR)
    {
        perror("connect()");
        return false;
    }

    return true;
```

Sendsocket() :

```
bool Liaisontcp::sendsocket(char* buffer)
{


    int send_socket = send(socket_to_send,buffer,14,0);


    if(send_socket == SOCKET_ERROR)
    {
        return false;
    }
    else  //la valeur de retour est la taille en octet
    {
        return true;
    }



}
```

Cette méthode prend en paramètre un pointeur de tableau de caractère (qui contient notre trame modbus).

Avec le fonction **send** nous envoyons le socket, la trame, la taille de celle-ci, et un flag que nous mettons à 0.

En cas de réussite, le serveur nous renvoie la taille en octet, on retourne vrai. (qui doit être de la même qu'à l'envoi)

En cas d'erreur, la taille est différente, on retourne donc faux.

Composante de la Trame :

```
char trame[14];

trame[0] = 0x00;
trame[1] = 0x00;
trame[2] = 0x00;
trame[3] = 0x00;
trame[4] = 0x00;
trame[5] = 0x06;
trame[6] = 0x00;
trame[7] = 0x06;
trame[8] = 0x00;
trame[9] = 0x02;
trame[10] = 0x00;
trame[11] = 0x12;
trame[12] = 0x01;
trame[13] = 0x01;

liaison->sendsocket(trame);
```

De la case 0 à 6 sont stockées des informations concernant l'encapsulation TCP :

L'indice 0 de notre tableau stocke l'identifiant de l'esclave. (allant de 1 à 247)

L'indice 2 de notre tableau stocke le numéro du protocole utilisé. (0 pour le protocole modbus/TCP)

L'indice 4 de notre tableau stocke la taille de notre message, étant donné que les mots clé font toujours 4 caractères donc cela correspond à 0x00.

L'indice 6 représente l'adresse xway.

De la case 7 à 13 sont stockées des informations concernant le protocole modbus en lui-même :

L'indice 7 stocke le code fonction de la trame, ici 0x06 correspond à l'écriture.

L’indice 8 stocke le mot clé qui va déclencher une cloche, ici 0x00 correspond à M2.0 donc la grosse cloche.

L’indice 10 stocke la valeur du mot.

L’indice 12 stocke le CRC, c’est une valeur calculée par l’émetteur (le maître) et le récepteur (l’esclave), si le CRC retourné est différent du CRC envoyé c’est qu’il y a une erreur.

L’indice 13 stocke la valeur binaire de la cloche que l’on veut faire sonner.

Par exemple si c’est la petite cloche : 0x03

2. Contraintes et problèmes rencontrés :

Nous avons perdu beaucoup de temps sur la gestion de notre matériel, il nous fallait un câble Ethernet suffisamment long pour relier la carte et notre machine, ensuite nous avons du, à l’aide de Mr Langlacé, rebrassé une prise réseau pour être sur le même réseau que la carte.

Il nous a été difficile de comprendre le passage théorique d’une trame modbus à un tableau de caractères stockant les différentes variables de la trame.

Conclusion

Pour améliorer notre travail, nous aurions pu mieux nous organiser et diviser les tâches, car nous faisions en général tous la même tâche en même temps..

Ce TP aura été très intéressant pour voir concrètement comment fonctionne une trame modbus encapsulée. Il est nécessaire de bien gérer son temps de travail pour le réussir.