Table des matières

Présentation

Le développement en C++ avec la liaison RS-232, également connue sous le nom de RS232, est une approche clé pour créer des applications qui communiquent avec des dispositifs série dans des environnements industriels et d'automatisation. La RS-232 est un protocole de communication série largement utilisé pour établir des connexions entre un ordinateur ou un système de contrôle et des dispositifs tels que des capteurs, des automates programmables industriels (API), des instruments de mesure, des imprimantes, etc.

La RS-232

La RS-232 est une norme de communication série qui définit les caractéristiques électriques et les signaux utilisés pour transmettre des données binaires entre deux équipements. Elle utilise des niveaux de tension positifs et négatifs pour représenter les bits de données. La RS-232 est simple à mettre en œuvre et offre une communication série asynchrone.

Au niveau industriel

La RS-232 est largement utilisée dans les environnements industriels pour plusieurs raisons :

- Elle offre une communication fiable et robuste pour la surveillance et le contrôle de dispositifs industriels.
- Elle est compatible avec de nombreux dispositifs industriels courants, ce qui en fait un choix privilégié pour l'automatisation industrielle.
- Elle est adaptée aux communications à courte distance, ce qui est fréquent dans les installations industrielles.

Envoyer des données sur le port série en langage python

Le programme suivant permet d'envoyer des données sur le port série COM1.

```python
import serial
import time

def envoyer_trame_rs232(port, baudrate, trame_ascii):
    try:
        # Ouvre le port série
        ser = serial.Serial(port, baudrate, timeout=1)

        # S'assure que le port est ouvert
        if ser.isOpen():
            print(f"Port {port} est ouvert avec un baudrate de {baudrate}.")

            # Convertir la trame ASCII en bytes
            trame_bytes = trame_ascii.encode('ascii')

            # Envoi de la trame
            ser.write(trame_bytes)
            print(f"Trame envoyée : {trame_ascii} (en ASCII)")

            # Attendre un peu pour s'assurer de la transmission complète
            time.sleep(1)

            # Fermer le port
            ser.close()
            print("Port série fermé.")
        else:
            print(f"Impossible d'ouvrir le port {port}.")
```

```
    except serial.SerialException as e:
        print(f"Erreur de communication : {e}")

if __name__ == "__main__":
    # Paramètres de la connexion RS232
    port = 'COM1 # Remplace par le port série approprié
    baudrate = 9600  # Ajuste la vitesse de transmission si nécessaire

    # La trame à envoyer (ASCII string)
    trame_ascii = 'BTS Ciel' # Exemple de trame en ASCII

    envoyer_trame_rs232(port, baudrate, trame_ascii)
```

Recevoir des données sur le port série en langage python

Le programme suivant permet de lire les informations reçues sur le port série COM1.

```
import serial
import time

def ecouter_port_rs232(port, baudrate, timeout=1):
    try:
        # Ouvrir le port série
        ser = serial.Serial(port, baudrate, timeout=timeout)

        # Vérifier si le port est ouvert
        if ser.isOpen():
            print(f"Port {port} est ouvert avec un baudrate de {baudrate}.")

            while True:
                # Lire les données disponibles dans le buffer
                trame = ser.read(ser.in_waiting)

                if trame:
                    print(f"Trame reçue : {trame}")

                # Ajouter un petit délai pour ne pas consommer trop de ressources CPU
                time.sleep(0.1)

        else:
            print(f"Impossible d'ouvrir le port {port}.")

    except serial.SerialException as e:
        print(f"Erreur de communication : {e}")
```

```
    finally:
        # Fermer le port série proprement
        ser.close()
        print("Port série fermé.")

if __name__ == "__main__":
    # Paramètres de la connexion RS232
    port = 'COM1'  # Remplace par le port série approprié
    baudrate = 9600  # Ajuste le baudrate si nécessaire

    # Écouter le port série pour recevoir des trames
    ecouter_port_rs232(port, baudrate)
```

Installation de python

Installer Python3 sur Windows puis installer la librairie qui permet d'utiliser le porte série :

pip3 install pyserial

Les fondamentaux

Lors du développement en C++ avec la RS-232, il est essentiel de comprendre les éléments fondamentaux suivants :

- Configuration des ports série :
  Les paramètres tels que la vitesse de transmission, les bits de données, les bits de stop et le contrôle de flux doivent être correctement configurés pour une communication RS-232 réussie.

- Ouverture et fermeture de ports :
  Vous devrez gérer l'ouverture et la fermeture des ports série pour établir et rompre des connexions avec les dispositifs.

- Lecture et écriture de données :
  Les opérations de lecture et d'écriture sont essentielles pour échanger des données avec les dispositifs série.

- Gestion des erreurs :
  La gestion des erreurs de communication, telles que les pertes de données, est cruciale dans les applications industrielles pour garantir la fiabilité des opérations.

Le C++ est un langage de programmation polyvalent qui offre des bibliothèques et des outils pour la programmation de la RS-232. En utilisant des bibliothèques appropriées, vous pouvez développer des applications C++ qui communiquent avec des dispositifs série dans des environnements industriels, facilitant ainsi le contrôle et la surveillance de divers processus industriels.

L'API Windows : il faut inclure le fichier « windows.h »

Le développeur en C++ sous Windows doit utiliser les éléments suivants fournis par l'API :
– une structure **DCB** pour initialiser la liaison série,
– une structure **COMSTAT** pour contrôler la liaison série,
– des **fonctions** pour initialiser la liaison série, lire et écrire des données.
De plus, le programmeur déclarera et utilisera des buffers d'émission et de réception des données série.
La documentation de l'API Windows est accessible sur Internet.

La structure DCB

Cette structure permet de définir les caractéristiques de la liaison série.

```
// The DCB structure defines the control setting for a serial communications device.

typedef struct _DCB { // dcb
          DWORD DCBlength; // sizeof(DCB)
          DWORD BaudRate; // current baud rate
          DWORD fBinary: 1; // binary mode, no EOF check
          DWORD fParity: 1; // enable parity checking
          DWORD fOutxCtsFlow:1; // CTS output flow control
          DWORD fOutxDsrFlow:1; // DSR output flow control
          DWORD fDtrControl:2; // DTR flow control type
          DWORD fDsrSensitivity:1; // DSR sensitivity
          DWORD fTXContinueOnXoff:1; // XOFF continues Tx
          DWORD fOutX: 1; // XON/XOFF out flow control
          DWORD fInX: 1; // XON/XOFF in flow control
          DWORD fErrorChar: 1; // enable error replacement
          DWORD fNull: 1; // enable null stripping
          DWORD fRtsControl:2; // RTS flow control
          DWORD fAbortOnError:1; // abort reads/writes on error
          DWORD fDummy2:17; // reserved
          WORD wReserved; // not currently used
          WORD XonLim; // transmit XON threshold
          WORD XoffLim; // transmit XOFF threshold
          BYTE ByteSize; // number of bits/byte, 4-8
          BYTE Parity; // 0-4=no,odd,even,mark,space
          BYTE StopBits; // 0,1,2 = 1, 1.5, 2
          char XonChar; // Tx and Rx XON character
          char XoffChar; // Tx and Rx XOFF character
          char ErrorChar; // error replacement character
          char EofChar; // end of input character
          char EvtChar; // received event character
          WORD wReserved1; // reserved; do not use
} DCB;
```

Nous allons seulement nous intéresser aux 4 champs suivants de la structure DCB (écrits en gras dans la déclaration de la structure).

```
DWORD BaudRate; // current baud rate
BYTE ByteSize; // number of bits/byte, 4-8
BYTE Parity; // 0-4=no,odd,even,mark,space
BYTE StopBits; // 0,1,2 = 1, 1.5, 2
```

La fonction CreateFile()

Cette fonction permet d'ouvrir un port de communication série.

Ci-dessous, un extrait de la documentation.

```
HANDLE CreateFile(
 LPCTSTR lpFileName, // pointer to name of the file
 DWORD dwDesiredAccess, // access (read-write) mode
 DWORD dwShareMode, // share mode
 LPSECURITY_ATTRIBUTES lpSecurityAttributes, // pointer to security attributes
 DWORD dwCreationDistribution, // how to create
 DWORD dwFlagsAndAttributes, // file attributes
 HANDLE hTemplateFile // handle to file with attributes to copy
 );
```

Il faut consulter la documentation complète pour comprendre tous les paramètres. L'API Windows redéfinit des types de données, comme :

HANDLE qui est l'équivalent d'un nombre entier,
LPCSTR qui est l'équivalent d'un pointeur sur une chaîne de caractères,
DWORD qui est l'équivalent d'un nombre entier de 32 bits.

Les ports série sont nommés sous Windows par les chaines "com1", "com2", "com3"…: la chaine "com" suivie du N° du port.

Un exemple permettant de simplifier les explications

```
// Déclarations
DCB StructureSerieWindows;
HANDLE hCom;
// Ouverture du port série « com1 » en lecture et écriture :
hCom= CreateFile("com1", GENERIC_READ|GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);
// traitement des erreurs
if( hCom==INVALID_HANDLE_VALUE) {
          cout<<"Erreur de création du port série"<<endl;
          …
}
// Si le port de communication a été ouvert avec succès, on initialise la liaison série à
// 9600 bits/s, 8 bits de données, pas de parité, 1 bit de stop:
// 1) Lire d'abord la structure DCB avec la fonction GetCommState
// 2) Modifier les 4 paramètres
// 3)Actualiser la structure DCB avec la fonction SetCommState
else
{
         GetCommState(hCom,&StructureSerieWindows); //lecture structure

         StructureSerieWindows.BaudRate=CBR_9600; //modification des 4 paramètres
         StructureSerieWindows.ByteSize=8;
         StructureSerieWindows.StopBits=ONESTOPBIT;
         StructureSerieWindows.Parity=NOPARITY;

         SetCommState(hCom,&StructureSerieWindows); //maj structure
}
```

La fonction GetCommState()

Cette fonction lit la structure DCB.

```
/*The GetCommState function fills in a device-control block (a DCB structure) with the current
control settings for a specified communications device.*/

BOOL GetCommState(
          HANDLE hFile, // handle of communications device
          LPDCB lpDCB // address of device-control block structure
);
```

La fonction SetCommState()

Cette fonction actualise la structure DCB.

```
/*The SetCommState function configures a communications device according to the specifications in a device-
control block (a DCB structure). The function reinitializes all hardware and control settings, but it does not
empty output or input queues.*/

BOOL SetCommState(
          HANDLE hFile, // handle of communications device
          LPDCB lpDCB // address of device-control block structure
);
```

La fonction PurgeComm()

Cette fonction vide les buffers d'émission et de réception.

```
/*The PurgeComm function can discard all characters from the output or input buffer of a specified
communications resource. It can also terminate pending read or write operations on the resource.*/

BOOL PurgeComm(
          HANDLE hFile, // handle of communications resource
          DWORD dwFlags // action to perform
);
```

La fonction WriteFile()

Cette fonction écrit des données dans le buffer d'émission.

```
/*The WriteFile function writes data to a file and is designed for both synchronous and asynchronous operation.
The function starts writing data to the file at the position indicated by the file pointer. After the write
operation has been completed, the file pointer is adjusted by the number of bytes actually written, except when
the file is opened with FILE_FLAG_OVERLAPPED. If the file handle was created for overlapped input and output
(I/O), the application must adjust the position of the file pointer after the write operation is finished.*/

BOOL WriteFile(
          HANDLE hFile, // handle to file to write to
          LPCVOID lpBuffer, // pointer to data to write to file
          DWORD nNumberOfBytesToWrite, // number of bytes to write
          LPDWORD lpNumberOfBytesWritten, // pointer to number of bytes written
          LPOVERLAPPED lpOverlapped // pointer to structure needed for overlapped I/O
);
```

Un exemple d'écriture dans le buffer d'émission

```
char *UnMessage= "Bonjour";

// émission série sur le port RS232
DWORD NbrCarEcrit;
bool teste;

PurgeComm(hCom, PURGE_TXCLEAR & PURGE_RXCLEAR);
teste=WriteFile(hCom, UnMessage , strlen(UnMessage), &NbrCarEcrit, NULL);

// traitement des erreurs
if(teste==false)
{
          cout<<"Erreur d'écriture sur le port série"<<endl;
          . . .
}
```

La fonction ReadFile()

Cette fonction lit les données dans le buffer de réception.

```
/*The ReadFile function reads data from a file, starting at the position indicated by the file pointer. After
the read operation has been completed, the file pointer is adjusted by the number of bytes actually read,
unless the file handle is created with the overlapped attribute. If the file handle is created for overlapped
input and output (I/O), the application must adjust the position of the file pointer after the read
operation.*/

BOOL ReadFile(
          HANDLE hFile, // handle of file to read
          LPVOID lpBuffer, // address of buffer that receives data
          DWORD nNumberOfBytesToRead, // number of bytes to read
          LPDWORD lpNumberOfBytesRead, // address of number of bytes read
          LPOVERLAPPED lpOverlapped // address of structure for data
);
```

La structure COMSTAT

Cette structure donne l'état de la liaison série.

```
/*The COMSTAT structure contains information about a communications device. This structure is filled by the
ClearCommError function.*/

typedef struct _COMSTAT {
          DWORD fCtsHold : 1; // Tx waiting for CTS signal
          DWORD fDsrHold : 1; // Tx waiting for DSR signal
          DWORD fRlsdHold : 1; // Tx waiting for RLSD signal
          DWORD fXoffHold : 1; // Tx waiting, XOFF char rec'd
          DWORD fXoffSent : 1; // Tx waiting, XOFF char sent
          DWORD fEof : 1; // EOF character sent
          DWORD fTxim : 1; // character waiting for Tx
          DWORD fReserved : 25; // reserved
          DWORD cbInQue; // bytes in input buffer
          DWORD cbOutQue; // bytes in output buffer
} COMSTAT, *LPCOMSTAT;
```

Cette structure est remplie en appelant la fonction **ClearCommError()**.
Le champ **cbInQue** contient le nombre d'octets pas encore lu dans le buffer de réception.

Un exemple de lecture du buffer de réception

```
COMSTAT cs;
DWORD NbrCarlus;
DWORD ErrorMask;
unsigned int n=0;
bool teste;

PurgeComm(hCom,PURGE_TXCLEAR & PURGE_RXCLEAR);
ClearCommError(hCom, &ErrorMask, &cs);

n= cs.cbInQue;
if(n >0) // on a reçu n données
{
          char *buff = new char[n];
          teste = ReadFile( hCom,buff,n,&NbrCarlus,NULL);
          if ( teste == false) {
          cout<<"Erreur de lecture sur le port série"<<endl;
          . . .
}
// Utilisation des données mises dans buff
. . .
// Fin de l'utilisation des données
delete(buff);
```

Lorsque le PC reçoit des données sur son port série, il remplit le buffer de réception. Il faudra que le programme de traitement vienne **lire régulièrement le buffer de réception**, car les données arrivent n'importe quand.
La structure **COMSTAT** permet de récupérer l'état de la liaison série, et dans le cas qui nous intéresse, c'est le nombre d'octets à lire dans le buffer de réception qu'il nous faut connaître, avant d'effectuer la lecture proprement dite :

```
n= cs.cbInQue;
```

n contient le nombre d'octets en attente de lecture dans le buffer de réception.
Pour lire la structure **COMSTAT**, nous utilisons la fonction **ClearCommError().**

La fonction ClearCommError ()

Cette fonction lit la structure **COMSTAT** et **place le résultat dans le champ lpStat.**

```
/*The ClearCommError function retrieves information about a communications error and reports the current status
of a communications device. The function is called when a communications error occurs, and it clears the
device's error flag to enable additional input and output (I/O) operations.*/
```

```
BOOL ClearCommError(
          HANDLE hFile, // handle to communications device
          LPDWORD lpErrors, // pointer to variable to receive error codes
          LPCOMSTAT lpStat // pointer to buffer for communications status
);
```

La lecture des données est obtenue finalement par la fonction **ReadFile()**, et on peut contrôler les erreurs de lecture grâce à la valeur booléenne retournée par la fonction **ReadFile()**.

La lecture des données reçues dans le buffer de réception doit être faite régulièrement, car les données arrivent quand elles veulent !

Activité : Programme de transmission de données

1. Effectuer la connexion physique entre les 2 ordinateurs en les reliant avec le câble fourni.

- Windows donne normalement le nom "com1" au port série physique de l'ordinateur.
- On peut vérifier par : Panneau de configuration (Afficher par : Petites icônes) > Système > Gestionnaire de périphériques et identifier le nom du port série actif.
- Utiliser sur les 2 ordinateurs l'application putty pour vérifier le fonctionnement.

2. Tester les programmes en python pour recevoir les données sur entre 2 ordinateurs.

3. Créer un programme en python qui émet et reçoit simultanément les informations qui transites par le port série.

4. Ecrire le programme récepteur qui affiche chaque caractère reçu. **Tester ce programme** avec Putty sur l'autre machine.

5. Ecrire le programme émetteur pour envoyer une chaine de caractères. **Tester ce programme** avec Putty sur l'autre machine.

6. Maintenant, vous allez **remplacer Putty** par un programme pour réceptionner ou émettre des données.

7. [BONUS] Organiser votre code avec des fonctions pour faciliter la lisibilité de celui-ci.