Developpez

Tutoriel

Version originale disponible sur http://lil-jam63.developpez.com
Auteur : Oujaber Mohamed (Lil_jam63)
Contact : lil_jam63@hotmail.com

Delphi

Analyse et utilisation d'une trame RC5

Ce tutoriel est une publication du site Developpez.com

1 - Introduction

A l'heure où nos micro-ordinateurs s'orientent littéralement vers le multimédia, bon nombre d'entres nous utilisent leurs PC pour la lecture de films, l'écoute de musique ou alors la visualisation de photo.

Pouvoir contrôler à distance cette « station multimédia » à l'aide d'une télécommande serait d'autant plus appréciable. Il existe pour cela des solutions ou des kits vendus en grande surface ou magasins spécialisés. Nous allons créer notre propre système de contrôle pour un moindre coût ce qui nous permettra d'avoir un contrôle total sur son fonctionnement.

2 - Moyens nécessaires

Pour réaliser notre « kit », nous aurons besoin de quelques fournitures :

- Une télécommande universelle (compatible RC5), (supportant la marque Philips) que l'on peux trouver dans n'importe quelle grande surface.
- Un récepteur infrarouge que l'on va faire nous même (pour un coût de 3-4 €) ou alors un récepteur IR universel sur port COM disponible dans le commerce ou sur internet.

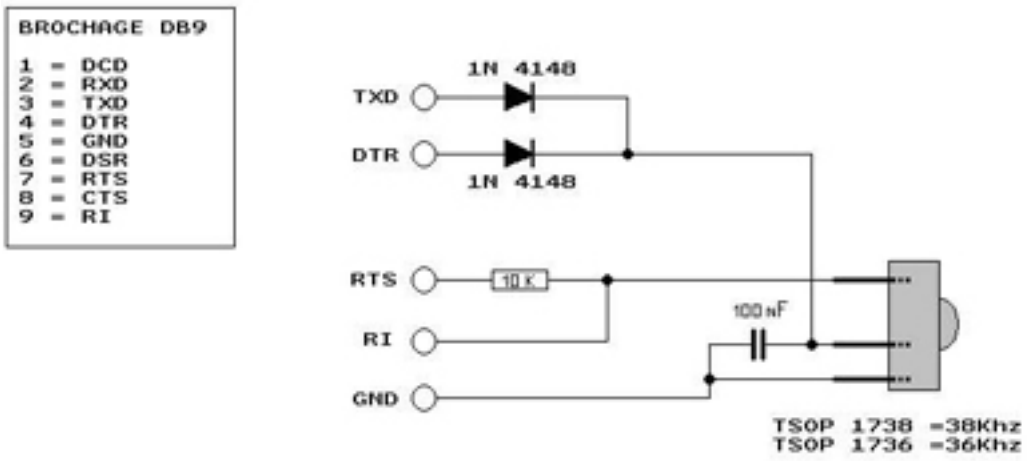
2.1 Solution "fer à souder"

Pour notre montage, nous avons besoins de certains composants électroniques que l'on trouve assez facilement car standard, je ne vais pas m'attarder sur cette partie du sujet qui ne nous intéresse pas vraiment ici.
Je vais simplement vous donner la liste des composants nécessaires ainsi que le schéma électrique (qui n'a rien de bien compliqué),
Donc voici la liste des composants nécessaires :

- 2 diodes 1N4148
- 1 résistance 10k
- 1 condensateur 100nF
- 1 récepteur IR TSOP1738

Maintenant le montage en lui-même, celui ci étant très simple, vous n'avez pas forcement besoin d'une plaque CI, vous pouvez le faire en fil volant par contre assurez vous quand même de ne pas provoquer de court-circuit en isolant les connections.

Note : Si vous utilisez un ordinateur portable, rajoutez un régulateur 78L05 en entrée. La sortie de certains ordinateurs portables n'étant pas assez régulée, vous risquez d'avoir des soucis d'alimentation.



2.2 Solution du commerce

Pour ceux qui ne sont pas à l'aide avec un fer à souder, vous pouvez également vous procurer des dispositifs un peu plus évolués dans le commerce ou sur internet, voici un exemple :

- http://lnx.manoweb.com/lirc/?partType=section&partName=buy (12€)

3 - Analyse du signal de la télécommande

3.1 Description

Pour pouvoir utiliser notre signal infrarouge, il faut abord savoir ce qu'il contient, le protocole utilisé.
Dans le cas des télécommandes infrarouge, 2 principaux protocoles sont utilisés :

- SIRCS, créé par Sony
- RC5, créé par Philips

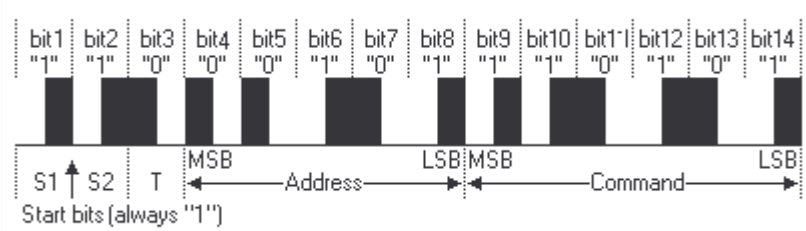
Mais d'autres existent comme le RC6, ITT, JVC, NEC, NRC17, SHARP IR, RECS80, …
Nous allons étudier le protocole RC5 que la majorité des télécommandes utilise.

3.2 Protocole RC5

Ce protocole est le plus répandu en Europe, la quasi-totalité des télécommandes universelles le supportent.
Une trame RC5 est composée de 14 bits classés en 4 groupes

- 2 bits de start
- 1 bit de répétition (ou commutation)
- 5 bits d'adressage
- 6 bits de commande

Les bits de start permettent d'informer le récepteur qu'une trame va être envoyée.
Le bit de répétition change d'état lorsque l'on relâche le bouton, cela va nous permettre de savoir si l'utilisateur reste appuyé ou pas sur le bouton. Utile lors d'une saisie de numéro (3 ou 33) ou d'un déplacement par exemple.
Les bits d'adressage désignent le périphérique pour lequel la trame est destinée (TV, VCR, SAT, …).
Puis les 6 derniers bits représentent la commande en elle-même.
Voilà donc à quoi ressemble une trame RC5 :



Comme vous pouvez le constater, les bits sont codés par fronts :
Front montant : bit = 1
Front descendant : bit = 0
C'est ce que l'on appelle le codage biphasé ou codage Manchester.
Chaque bit a une longueur de 1,778ms, ce qui fait qu'une trame complète fait 14*1,778 = 24,892ms mais officiellement la durée est de 24,889ms.
Le temps entre chaque trame est de 113,78ms.

3.3 Mise en œuvre

Ouvrez Delphi et créez un nouveau projet, puis enregistrer le sous « TestRC5.dpr » tout en ayant enregistrer l'unité principale sous « uMain.pas ».
Ensuite pour bien structurer notre projet et séparer les parties, nous allons rajouter une unité que l'on va appeler « uThread.pas » et une autre « uUtils.pas ».

Note : Vous pouvez également générer le thread en passant par l'assistant :
Fichier->Nouveau->Autre…->Objet Thread -> nom de classe : TThreadRC5

L'unité principale « uMain.pas » regroupera la gestion générale de l'application, « uThread.pas » la gestion du thread et l'analyse du signal et enfin « uUtils.pas » comme son nom l'indique, les différentes fonctions, procedures et types nécessaires.
A cette fiche, nous allons rajouter des contrôles qui vont nous permettre d'afficher les résultats de cette analyse. Reproduisez la fiche ci-contre dans votre projet tout en prenant soin de nommez vos TEdits de la même manière et le TComboBox **cbComPort**, les boutons Démarrer et Arreter s'appelleront respectivement **btnStart** et **btnStop**.

Ceci étant fait, passons à l'unité « uUtils.pas ».
Commencez par rajouter ceci dans les **uses** pour la suite

```
uses
  Windows, Messages;
```

Pour notre projet, nous allons avoir besoin d'utiliser un message WM_COPYDATA pour communiquer de notre thread vers notre fiche. Ce message nous permet contrairement aux autres, d'échanger une structure ou plutôt un pointeur vers une structure.

Il prend en paramètre un pointeur sur une structure COPYDATASTRUCT

```
typedef struct tagCOPYDATASTRUCT {
    ULONG_PTR dwData;
    DWORD cbData;
    PVOID lpData;
} COPYDATASTRUCT, *PCOPYDATASTRUCT;
```

que nous allons déclarer de cette manière :

```
  { Structure utilisée par le message WM_COPYDATA  }
  PCopyData = ^TCopyData;
  TCopyData = record
      dwData: LongInt;
      cbData: LongInt;
      lpData: Pointer;
    end;
```

Une deuxième structure nous sera nécessaire pour encapsuler les différents résultats fournis par notre analyse, déclarez là de cette manière :

```
  { Structure qui va contenir nos infos  }
  PRC5Data = ^TRC5Data;
  TRC5Data = record
      Code          : string;   // représentation binaire de la trame
      Adresse       : string;   // l'adresse du récepteur pour lequel est destiné la
trame
      Commande      : string;   // la commande
      BitControl    : integer;  // l'état du bit de répétition
      CommandeHexa  : string;   // représentation hexadécimale de la commande
    end;
```

La configuration du port COM requiert également 3 énumérations pour la parité, le bit de stop ainsi que le mode de réception utilisé par la structure _DCB.
Rajoutez donc ceci :

```
  { Enumération port COM  }
  TParity = ( ptNONE, ptODD, ptEVEN, ptMARK, ptSPACE );
  TStopBit = ( sbONE, sbONE5, sbTWO );
  TReceiveMode = ( rmRAW, rmTERM );
```

A ce stade du tutorial, votre unité “uUtils.pas” doit ressembler à ceci

```
unit uUtils;

interface

uses
  Windows, Messages;

type
  { Enumération port COM  }
  TParity = ( ptNONE, ptODD, ptEVEN, ptMARK, ptSPACE );
  TStopBit = ( sbONE, sbONE5, snTWO );
  TReceiveMode = ( rmRAW, rmTERM );

  { Structure utilisée par le message WM_COPYDATA  }
```

```
  PCopyData = ^TCopyData;
  TCopyData = record
      dwData: LongInt;
      cbData: LongInt;
      lpData: Pointer;
    end;

  { Structure qui va contenir nos infos  }
  PRC5Data = ^TRC5Data;
  TRC5Data = record
      Code          : string;   // représentation binaire de la trame
      Adresse       : string;   // l'adresse du récepteur pour lequel est destiné la
trame
      Commande      : string;   // la commande
      BitControl    : integer;  // l'état du bit de répétition
      CommandeHexa  : string;   // représentation hexadécimale de la commande
    end;

implementation

end .
```

Retournons à l'unité « uThread.pas » pour nous occuper du thread.

3.3.1 Thread

Rajoutez ceci dans les **uses**

```
uses
  Classes, Windows, SysUtils, StrUtils, uUtils, Messages;
```

L'utilisation d'un thread dans notre cas nous permet de travailler en tache de fond sans monopoliser les ressources système et surtout de ne pas geler l'application.
Pour la partie analyse, nous aurons besoin d'une surveillance permanente su port COM ce qui aura pour effet de geler l'application et celle ci ne sera plus rafraîchit. 2 méthodes sont à notre disposition pour utiliser les thread, la première fournie par windows est l'appel à la fonction createThread() et l'autre est la classe TThread fournie par Delphi, pour des raisons de facilité de mise en œuvre nous utiliserons la classe TThread .Rajoutez la déclaration de notre thread ou alors rajoutez les termes manquant si vous avez utilisé l'assistant.

```
TThreadRC5 = class(TThread)
    private
      { Déclarations privées }
      bin: string;                // représentation binaire de la trame
      RC5Data: TRC5Data;          // structure qui va contenir nos infos sur la trame
      CopyData: TCopyData;        // structure qui va être envoyé avec le message
WM_COPYDATA et qui va contenir un pointeur sur la structure TRC5data
      hHandle: THandle;           // handle de la fiche qui réceptionne le message
WM_COPYDATA
      hCom: THandle;              // handle du port COM
    protected
      { Déclarations protégées }
      procedure Analyse;          // procedure de surveillance du port COM
      procedure Send;             // va envoyer le message WM_COPYDATA avec ses infos
    public
      { Déclarations publiques }
      constructor Create(AHandle, ACom: THandle);
      procedure Execute; override;
    end;
```

Comme vous pouvez le voir, celui çi est plutot « simple », le plus gros du travail s'effectuera dans la procedure « **Analyse »** qui est chargée de la surveillance du port COM et de l'analyse que nous aborderons plus tard. Pour des raisons de commodité, et de réutilisation, le constructeur du thread va prendre en paramètre le handle du port COM et celui de la fiche que l'on doit avertir. La procedure « **Send »** quand à elle va se charger d'envoyer les données à la fiche qui réagira en conséquence.
Passons au constructeur du thread, rajoutez ceci

```
constructor TThreadRC5.Create(AHandle, ACom: THandle);
begin
  inherited Create(False);  // execution du thread lors de sa création
  hHandle := AHandle;       // handle de la fiche Form1 (pour le message WM_COPYDATA)
  hCom := ACom;             // handle du port COM

  FreeOnTerminate := True;
  priority := tpHigher;
end;
```

Lors de sa création, nous allons appeler le constructeur grâce à **inherited** avec comme paramètre **false** qui a pour conséquence de le démarrer dès sa création (appel de la méthode **Execute**). Ensuite on initialise les variables **hHandle** et **hCom** avec respectivement le handle de la fiche et celui du port COM.
Nous lui donnons également une priorité supérieure et lui demandons de se terminer lors de sa destruction.

La procédure **Execute**, appelée lors de la création du thread va simplement se contenter d'appeler notre procédure de surveillance et de décodage « **Analyse** »

```
procedure TThreadRC5.Execute;
begin
  while not Terminated do
    begin
      Analyse;                                    // on lance la surveillance
    end;
end;
```

Nous y arrivons enfin, la partie réception et analyse du signal.
Avant de commencer, nous allons avoir besoin d'une minuterie assez performante et précise.
Pour celà, le système met à notre disposition 4 méthodes de mesure du temps qui sont

- **Now, Time, Timer**
- **GetTickCount()**
- **TimeGetTime()**
- **QueryPerformanceCounter()**

Voilà un tableau comparatif issue de chez Microsoft concernant la précision de chacun :

Function	Units	Resolution
Now, Time, Timer	seconds	1 second
GetTickCount	milliseconds	approx. 10 ms
TimeGetTime	milliseconds	approx. 10 ms
QueryPerformanceCounter	QueryPerformanceFrequency	same

Pour notre analyse, nous avons besoin d'une précision de l'ordre de la microseconde (µs),
nous pouvons d'ores et déjà supprimer les 3 premiers car il ne sont pas assez performant, le meilleur choix se situe donc dans QueryPerformanceCounter() car sa précision est basée sur la fréquence du processeur.

Ne pouvant pas travailler avec les bits directement, nous allons réagir aux changements d'état de DSR tous les ¾ de bit, c'est-à-dire juste après le front (montant ou descendant) pour en déterminer l'état.
QueryPerformanceFrequency() va nous permettre de récupérer la fréquence qui va nous servir à déterminer le temps écoulé grâce à cette formule :

T = (t1-t0)*précision / fréquence

La précision sera de 1000000 pour avoir une précision à la microseconde (1s = 1000000 µs).
Pour déclencher le chrono, nous allons également utiliser WaitCommEvent() qui va nous permettre d'attendre un événement sur DSR grâce au flag EV_DSR .

Voilà donc comment va se présenter notre procedure « **Analyse ».**

```
procedure TThreadRC5.Analyse;
var
  i, j, tps : integer;
        bBits : boolean;
        nbCom : integer;
  Errors : DWord;
        PerformanceCount, PerformanceCountOld, Frequency : TLargeInteger;
  szCom : array [0..256] of char;
  szComOld : array [0..256] of char;
begin
  i := 2;       // les 2 premières impulsions du bit de start
  j := 0;       // index des tableaux de bits
  nbCom := 0;   // nombre d'évenements sur ligne DSR

  SetCommMask(hCom,EV_DSR);                          // on applique un masque sur le
changement d'état de DSR
  QueryPerformanceFrequency(Frequency);              // on récupère la fréquence du
compteur haute performance
  while not Terminated do                            // tant que le thread est activé
    begin
      WaitCommEvent(hCom,Errors,nil);                // on attend les évenements DSR
      inc(nbCom);                                    // on incrémente le nombre
d'impulsions (événements)
      PerformanceCountOld := PerformanceCount;
      QueryPerformanceCounter(PerformanceCount);  // on récupère la valeur courante du
compteur haute performance
      Tps := Trunc((PerformanceCount-PerformanceCountOld)*1000000/Frequency); // temps
écoulé en µs: (t1 - t0) * 1000000 / frequence.  (1s = 1000000µs)

      if (Tps > 13335)then                           // si Tps > à la longueur de 3/4 de
bits * 10 (1 bits = 1778 µs --> 1778*0.75 = 1333.5 * 10 = 13335)
      begin
        nbCom :=1;                                   // on réinitialise à 1
        i :=2;                                       // toujours notre bit de start
        j :=0;                                       // l'index de tableau à 0
        FillChar(szCom, sizeof(szCom),0);            // on remplit notre tableau de bits
actuel de 0
        FillChar(szComOld, sizeof(szComOld),0);   // on remplit notre ancien tableau
de bits de 0
      end;

      if nbCom = i then                              // si on as notre bit de start ou
qu'il n'y a pas eu de changement d'état pour le bit précédent
      begin
        bBits :=(nbCom mod 2 =0);                    // on vérifie si nbCom est un
multiple de 2
```

```
        if bBits then szCom[j]:= chr($31) else  szCom[j]:= chr($30);  // si oui alors
on met le bit à 1 sinon 0
        inc(j);                                    // on incremente l'index du tableau
        if(Tps > 1333) then                        // si Tps > à la longueur de 3/4 de
bits c'est qu'il n'y a pas eu de changement d'état
          begin
            if not bBits then szCom[j]:= chr($31) else  szCom[j]:= chr($30); // et que
ce n'est pas un multiple de 2 alors 1 sinon 0
            inc(i);
          end
        else
          begin
            inc(i,2);                               // on incremente de 2
            if not bBits then szCom[j]:= chr($30); // si Tps est inférieur à 3/4 de
bits et non multiple de 2 alors 0
          end;
      end;

      // si on a nos 14 bits et que notre tableau est différent du précédent
      if (strlen(szCom)=14) and (strcomp(szCom,szComOld)<>0) then
        begin
          strcopy(szComOld,szCom);  // on sauvegarde notre tableau de bits
          bin := string(szComOld);  // on récupère notre représentation binaire
          synchronize(Send);        // et enfin on déclenche l'évenement de réception
          Sleep(5);                 // une petite pause de 5 ms
        end;

    end;
end;
```

Donc une fois nos variables initialisées et le masque activé, on entre dans un boucle infinie qui se termine avec l'arrêt du thread puis on attend les événements DSR, lorsqu'un événement surgit, on récupère la valeur du compteur haute performance et on applique la formule citée précédemment pour obtenir le temps écoulé depuis le dernier événement :

T = (t1-t0)*précision / fréquence

Si celui-ci est supérieur à 10*3/4 de bit (13335µs), on estime que c'est une nouvelle trame qui commence (rappel : le protocole RC5 prévoit 113,78ms entre 2 trames) et on réinitialise nos variables sinon on teste si on a nos 2 bits de start auquel cas on commence l'analyse.
L'analyse proprement dite se fait ici :

```
if nbCom = i then
      begin
        bBits :=(nbCom mod 2 =0);
        if bBits then szCom[j]:= chr($31) else  szCom[j]:= chr($30);
        inc(j);
        if(Tps > 1333) then
          begin
            if not bBits then szCom[j]:= chr($31) else  szCom[j]:= chr($30);
            inc(i);
          end
        else
          begin
            inc(i,2);
            if not bBits then szCom[j]:= chr($30);
          end;
      end;
```

Cette partie étant assez compliquée à comprendre, je vais la détaillée avec un exemple.
On part du principe que le 1er bit de start est toujours à 1 ce qui est le cas alors prenons l'exemple de la touche 1 (11 0 00000 000001) :
Quand on a nos 2 premières impulsions, le 1er bit est passé donc on rentre dans la condition nbCom= i car

```
nbCom = 2       -> valeur incrémentée par le nombre d'impulsions (2 par bit)
i = 2           -> valeur d'initialisation
```

et puisque 2 mod 2 = 0, on rentre également dans la condition bBits et on met le premier bit à 1

```
chr($31) = 1    -> 31 est la valeur hexadécimale de 1
```

ensuite on incrémente l'index du tableau pour passer au 2ème bit de start, si **Tps** est supérieur à ¾ de bit (1333.5µs) c'est qu'il n'y a pas eu de changement d'état donc on incrémente **i** de 2 et comme **bBits** est true, on ne touche pas au tableau.
Lors de l'impulsion suivante, on incrémente **nbCom** qui prend la valeur de 3 mais on ne rentre plus dans notre condition car

```
nbCom = 3       -> incrémentée lors de la dernière impulsion
i = 4           -> incrémentée de 2 lors du dernier test
```

donc on continue notre cycle jusqu'à la prochaine impulsion qui incrémente **nbCom** et nous fait revenir à nbCom = i et on continue les tests, 4 mod 2 = 0 donc on met le bit 2 à 1 également, voilà nos 2 bits de start.
On incrémente l'index du tableau pour passer au bit de répétition. Le bit de répétition étant par défaut à 0, il y a changement d'état ce qui sera valider par **Tps** qui sera inférieur à 1333µs et donc **bBits** étant true, on met le bit à 0 et on incrémente **i**.
J'espère que vous me suivez jusque là, ce qu'il faut retenir c'est que tout est basé (déterminé) par rapport à l'espace entre les impulsions en partant du principe que le premier bit est à 1.
On continue, on passe aux bits d'adresse, nouvelle impulsion donc incrémentation de **nbCom** et à nouveau égalité avec **i**

```
nbCom = 5
i = 5
```

5 mod 2 <> 0 donc on met le bit de répétition à 0 ce qui avait déjà été fait par le test d'avant et on incrémente pour passer au 1er bit d'adresse, toujours pareil donc si un changement d'état à été fait, le temps est inférieur à 1333µs donc on met le bit à 1 car **bBits** est true sinon on le met à 0 comme dans ce cas-ci puis on incrémente **i** et on continue le cycle.
Ainsi de suite jusqu'à ce que notre tableau fasse 14 bits

```
      if (strlen(szCom)=14) and (strcomp(szCom,szComOld)<>0) then
        begin
          strcopy(szComOld,szCom);  // on sauvegarde notre tableau de bits
          bin := string(szComOld);  // on récupère notre représentation binaire
          synchronize(Send);        // et enfin on déclenche l'évenement de réception
          Sleep(5);                 // une petite pause de 5 ms
        end;
```

Dès que notre tableau de bit fait 14, on le sauvegarde et on appelle la procedure « **Send** »
La petite pause n'est pas nécessaire mais ne mange pas pain non plus donc autant en mettre une pour laisser le thread respirer un peu, son absence n'affecte en rien l'efficacité de la réception.

Passons maintenant à la partie communication entre notre thread et notre fiche principale, on aurait pu utiliser les appels en dur dans notre thread (Form1. ???) mais cela ne favorise pas la réutilisation du code,

autant utiliser un système qui ne va pas nous gêner si l'on souhaite utiliser ce thread dans une autre application.

Note : Le plus simple est de faire un composant qui l'intègre mais ce n'est pas le but de ce tutorial, sjrd (sjrd.developpez.com) vous en propose un complet à ce sujet.

Cette tâche est assurée par la procédure **Send** de notre thread qui se présente ainsi

```
procedure TThreadRC5.Send;
begin
  { Structure TRC5Data }
  with RC5Data do
    begin
      Code := bin;                                    // représentation binaire
      Adresse := AnsiMidStr(bin,4,5);                 // extraction de l'adresse
      Commande := AnsiMidStr(bin,9,6);                // extraction de la commande
      BitControl := strToInt(bin[3]);                 // extraction du bit de
répétition
      CommandeHexa := BinToHex(AnsiMidStr(bin,9,6));  // conversion de la commande
sous sa forme hexadécimale
    end;
  { Structure TCopyData }
  with CopyData do
    begin
     dwData := 1;
     cbData := sizeof(RC5Data);
     lpData := @RC5Data;                              // pointeur sur la structure
TRC5Data
    end;

  // on envoie le message WM_COPYDATA avec un pointeur sur notre structure TRC5Data en
paramètre
  SendMessage( hHandle, WM_COPYDATA, 0, LongInt(@CopyData));

end;
```

On y retrouve nos deux structures **TRC5Data** et **TCopyData**. Avant d'envoyer quoi que ce soit, on remplit la structure **RC5Data** avec les informations obtenus par l'analyse donc la trame binaire complète, les bits d'adresse, les bits de commande ainsi que l'état du bit de répétition et la représentation hexadécimale des bits de commande que l'on obtiendra grâce à la fonction **binToHex()** que l'on va rajouter dans l'unité « uUtils.pas »

```
function BinToHex(BinStr: string): string;
const
  BinArray: array[0..15, 0..1] of string =
    (('0000', '0'), ('0001', '1'), ('0010', '2'), ('0011', '3'),
     ('0100', '4'), ('0101', '5'), ('0110', '6'), ('0111', '7'),
     ('1000', '8'), ('1001', '9'), ('1010', 'A'), ('1011', 'B'),
     ('1100', 'C'), ('1101', 'D'), ('1110', 'E'), ('1111', 'F'));
var
  Error: Boolean;
  j: Integer;
  BinPart: string;
begin
  Result:='';

  Error:=False;
  for j:=1 to Length(BinStr) do
    if not (BinStr[j] in ['0', '1']) then
```

```
      begin
        Error:=True;
        Break;
      end;

  if not Error then
    begin
      case Length(BinStr) mod 4 of
          1: BinStr:='000'+BinStr;
          2: BinStr:='00'+BinStr;
          3: BinStr:='0'+BinStr;
        end;

      while Length(BinStr)>0 do
        begin
          BinPart:=Copy(BinStr, Length(BinStr)-3, 4);
          Delete(BinStr, Length(BinStr)-3, 4);
          for j:=1 to 16 do
            if BinPart=BinArray[j-1, 0] then
              Result:=BinArray[j-1, 1]+Result;
        end;
    end;
end;
```

Ainsi que sa déclaration dans l'interface

```
function BinToHex(BinStr: string): string;
```

Ensuite, c'est la structure **CopyData** que l'on va envoyer en paramètre du message WM_COPYDATA qui elle aussi va prendre en paramètre **lpData** un pointeur sur la structure **RC5Data**.
Le message est envoyé en utilisant sendMessage(), **hHandle** est le handle de notre fenêtre principale fournie au thread lors de l'appel de son constructeur. Cette fonction attend en quatrième paramètre un entier on doit donc transtyper notre pointeur en **longInt**.

Voilà pour le thread, maintenant passons à la fiche principale.

3.3.2 Fiche

Pour commencer il va falloir que l'on intercepte le message WM_COPYDATA envoyé par le thread donc rajoutez cette procedure dans la section private de votre fiche

```
procedure Receive(var msg: TMessage); message WM_COPYDATA;
```

et son implementation

```
procedure TForm1.Receive(var msg: TMessage);
begin
  // Quand on reçois le message de réception, on décortique le message pour extraire
notre structure qui contient les infos sur la trame
  with PRC5Data(PCopyData(msg.LParam)^.lpData)^ do
    begin
      edTrameBinaire.Text := Code;            // représentation binaire de la trame
      edBitRep.Text := intToStr(BitControl);  // l'état du bit de répétition
      edAddress.Text := Adresse;              // l'adresse du récepteur pour lequel
était destiné cette trame
      edCommande.Text := Commande;            // la commande
      edComHex.Text := CommandeHexa;          // représentation hexadécimale de la
commande
```

```
    end;
end;
```

Cette procédure est déclenchée à chaque réception du message WM_COPYDATA, pour extraire les données on inverse le transtypage effectué lors l'envoie du message.

```
with PRC5Data(PCopyData(msg.LParam)^.lpData)^ do
```

On transtype le champ LParam dans le type PCopyData puis le champ lpData en type PRC5Data.
Il ne reste plus qu'à afficher nos données dans les TEdits prévus à cet effet.
Tout cela ne sera pas possible qu'après avoir ouvert au préalable une communication sur le port COM.
Windows met à notre disposition createFile() qui permet également d'ouvrir un port COM. Cette communication nécessitera également une structure **_DCB** qui servira à configurer notre port, donc rajoutez ceci dans la section **private** de la fiche

```
dcb : _DCB;
```

Voici l'implementation de l'évenement **onClick()** du bouton **btnStart**

```
procedure TForm1.btnStartClick(Sender: TObject);
begin
  { Ouverture du port COM }
  hcom := CreateFile(pChar(cbComPort.Text), GENERIC_READ, 0, nil, OPEN_EXISTING,
FILE_FLAG_OVERLAPPED, 0);

  { Configuration du port COM  }
  dcb.BaudRate := 9600;
  dcb.StopBits := DWord(sbOne) ;
  dcb.Parity := DWord(ptNone) ;
  dcb.ByteSize := 8 ;

  if (hcom = INVALID_HANDLE_VALUE) then
    begin
      // Si l'ouverture à échouée, on affiche le message d'erreur et on quitte la
procedure
      Application.MessageBox(PChar(SysErrorMessage(GetLastError())),'');
      exit;
    end
  else StatusBar.Panels[0].Text := 'Connecté';

  { Application de la configuration au port COM }
  if not SetCommState(hcom,dcb) then StatusBar.Panels[0].Text := 'Erreur lors de
l''initialisation';

  SetDTR(hCom, true);                          // On met la ligne DTR à 1 (Suivant les
récepteurs)
  setRTS(hCom, true);                          // On met la ligne RTS à 1 (Suivant les
récepteurs)

  //Et enfin on lance notre thread qui va surveiller ce qui arrive sur le port COM
  Thread := TThreadRC5.Create(Handle, hCom);
end;
```

On ouvre la communication sur le port COM sélectionné dans le comboBox **cbComPort**,
A l'aide de la structure _DCB, on configure celui-ci. **sbOne** et **ptNone** font partie respectivement des énumérations **TParity**, **TStopBit** déjà déclarés dans l'unité « uUtils.pas ». Certains récepteurs comme celui représenté par le schéma en première partie, nécessite la mise à 1 des broches DTR et RTS ou l'une d'entres

elles. Pour cela nous allons ajouter dans la partie interface de l'unité « uUtils.pas » les procédures **setDTR()** et **setRTS()**, les déclarations à rajouter dans la partie interface

```
procedure SetDTR( hCom: THandle; Status: boolean );
procedure SetRTS( hCom: THandle; Status: boolean );
```

Et leurs implementations:

```
procedure SetDTR( hCom: THandle; Status: boolean );
begin
  case Status of
    true:   EscapeCommFunction( hCom, Windows.SETDTR );    // Mise à 1
    false:  EscapeCommFunction( hCom, Windows.CLRDTR );    // Mise à 0
  end;
end;

procedure SetRTS( hCom: THandle; Status: boolean );
begin
  case Status of
    true:   EscapeCommFunction( hCom, Windows.SETRTS );    // Mise à 1
    false:  EscapeCommFunction( hCom, Windows.CLRRTS );    // Mise à 0
  end;
end;
```

Elles utilisent toutes deux la function EscapeCommFunction() mise à disposition par windows.

Une fois la communication ouverte et configurée sans erreur, on appel le thread en lui passant en paramètre le handle de notre fiche et celui du port COM

```
Thread := TThreadRC5.Create(Handle, hCom);
```

Dans l'événement **onClick()** du bouton **btnStop**, on va terminer le thread et fermer le port COM, voici son implémentation

```
procedure TForm1.btnStopClick(Sender: TObject);
begin
  // Si le thread est déjà lançé, on le termine
  if Thread <> nil then Thread.Terminate;
  CloseHandle(hCom);                          // on ferme la connection au port COM
  StatusBar.Panels[0].Text := 'Déconnecté';
end;
```

3.4 Exemple d'utilisation

Les exemples d'utilisation ne manquent pas, vous pouvez affecter n'importe quelle action à une touche prédéfinie, il vous suffit simplement de tester la valeur du code touche renvoyé par le thread. Je vous en propose 2 qui peuvent être intéressant.

3.4.1 Piloter le lecteur Windows Media Player

Pour illustrer l'intérêt d'un tel système, nous allons piloter le lecteur multimédia Windows Media Player.

Ce qui vous permettra de visionner vos vidéos ou alors écouter vos morceaux de musique tout en étant confortablement assis sans avoir à vous déplacer pour monter le son, mettre en plein écran, avancer dans votre sélection, etc.…
Tout d'abord pour communiquer avec notre lecteur multimédia, nous allons utiliser la simulation de touche, c'est le système le plus simple à mettre en œuvre et facilement adaptable pour un autre lecteur.
Dans cet exemple, nous n'aborderons que les commandes Lecture, Arret, Plein écran, Silence pour cela, nous allons déclarer un type énumerateur **TWMPCommand** dans la section **type** de « uMain.pas » comme ceci :

```
TWMPCommand = ( wmpTogglePlay, wmpStop, wmpToggleScreen, wmpToggleMute);
```

Qui correspondent aux commandes suivantes:

- wmpTogglePlay → Lecture
- wmpStop → Arret
- wmpToggleScreen → Activation ou désactivation du plein écran
- wmpToggleMute → Activation ou désactivation du son

On rajoute également à notre projet un procédure qui va gérer la communication avec le lecteur que l'on va nommer **RemotePlayer()**, rajouter donc sa déclaration dans la section **Public** de « uMain.pas » de cette manière

```
procedure RemoteWMP(commande: TWMPCommand);
```

ensuite viens son implementation

```
{=============================================================================}
{   Commande de Windows Media Player                                          }
{=============================================================================}
procedure TForm1.RemotePlayer(commande: TWMPCommand);
var
  hMediaPlayer: HWND;
  hWinForeground: HWND;
begin
  if hMediaPlayer = 0 then FindWindow('WMPlayerApp', nil);
  if hMediaPlayer <> 0 then
    begin
      hWinForeground := getForegroundWindow();
      if hWinForeground <> hMediaPlayer then
        setForeGroundWindow(hMediaPlayer);
      sleep(10);
      case commande of
          wmpTogglePlay:
            begin
              keybd_event(VK_CONTROL, 0, 0, 0);
              keybd_event(80, 0, 0, 0);
              keybd_event(80, 0, KEYEVENTF_KEYUP, 0);
              keybd_event(VK_CONTROL, 0, KEYEVENTF_KEYUP, 0);
            end;
          wmpStop:
            begin
              keybd_event(VK_CONTROL, 0, 0, 0);
              keybd_event(83, 0, 0, 0);
              keybd_event(83, 0, KEYEVENTF_KEYUP, 0);
              keybd_event(VK_CONTROL, 0, KEYEVENTF_KEYUP, 0);
            end;
          wmpToggleScreen:
            begin
              keybd_event(VK_MENU, 0, 0, 0);
```

```
              keybd_event(VK_RETURN, 0, 0, 0);
              keybd_event(VK_RETURN, 0, KEYEVENTF_KEYUP, 0);
              keybd_event(VK_MENU, 0, KEYEVENTF_KEYUP, 0);
            end;
          wmpToggleMute:
            begin
              keybd_event(VK_F8, 0, 0, 0);
              keybd_event(VK_F8, 0, KEYEVENTF_KEYUP, 0);
            end;
        end;
    end;
end;
```

Cette procédure prend en paramètre le type commande à envoyer au lecteur, avant tout l'on recherche la fenêtre du lecteur en utilisant FindWindow(), **WMPlayerApp** est le nom de la classe de la fenêtre du lecteur, si celle çi a été trouvée, on la passe au premier plan pour s'assurer que ce sera bien elle qui aura le focus au moment de la simulation des touches clavier en utilisant setForeGroundWindow(). Une fois le lecteur en premier plan, en fonction de la commande passée en paramètre, on simule les touches clavier correspondantes à l'aide de keydb_event() lequel prend en paramètre le code touche ainsi que son état, dans notre exemple, c'est le raccourcis CTRL et P qui déclenche la lecture **VK_CONTROL** étant la constante définie de la touche CTRL et **80**, l'équivalent ASCII de la touche P, le troisième paramètre de keydb_event() indique l'état de la touche, **0** étant l'état appuyée, **KEYEVENTF_KEYUP** l'état relâchée. Vous remarquerez également que l'on simule d'abord l'appui sur la touche CTRL, ensuite l'appui sur la touche P et finalement le relâchement des 2 touches.

Note : Vous trouverez en annexe, un lien vers les différents codes clavier.

Rien ne vous empêche de rajouter des commandes en complétant le type **TWMPCommand**, les raccourcis affectés à chaque actions sont généralement indiquées dans le menu du lecteur ou alors dans les paramètres de ce dernier.
Maintenant nous devons affecter une touche de la télécommande à chacune de nos actions, pour faire simple, nous allons choisir les touches 1, 2, 3, 4.Rajoutez ces quelques lignes à la procedure **Receive()**

```
procedure TForm1.Receive(var msg: TMessage);
begin
  // Quand on reçois le message de réception, on décortique le message pour extraire
notre structure qui contient les infos sur la trame
  with PRC5Data(PCopyData(msg.LParam)^.lpData)^ do
    begin
      edTrameBinaire.Text := Code;            // représentation binaire de la trame
      edBitRep.Text := intToStr(BitControl);  // l'état du bit de répétition
      edAddress.Text := Adresse;              // l'adresse du récepteur pour lequel
était destiné cette trame
      edCommande.Text := Commande;            // la commande
      edComHex.Text := CommandeHexa;          // représentation hexadécimale de la
commande

      if (CommandeHexa = '01') then RemotePlayer(wmpTogglePlay)        // touche 1
      else if (CommandeHexa = '02') then RemotePlayer(wmpStop)         // touche 2
      else if (CommandeHexa = '03') then RemotePlayer(wmpToggleScreen) // touche 3
      else if (CommandeHexa = '04') then RemotePlayer(wmpToggleMute);  // touche 4

    end;
end;
```

La propriété **CommandeHexa** nous renvoie le code correspondant à la touche télécommande sélectionnée, on teste si une action est associée à ce code l'on appelle la procédure **RemotePlayer()** avec les paramètres nécessaires.

Note : Vous trouverez en annexe, une liste des codes touches télécommande ainsi que les adresses des pricipaux périphériques (TV, Magnetoscope, …).

Vous pouvez facilement adapter cette procédure à un autre lecteur multimédia, en modifiant le paramètre classname de FindWindow() et les combinaisons de touches. Voilà une liste de className des principaux lecteurs multimedia :

- BsPlayer → BSPlayer
- CoolPlayer → CoolPlayer
- DivXPlayer → Qwidget
- MoreTv → MoreTV
- PowerDVD → class of CyberLink Universal Player
- Winamp3 → Studio
- WinDVD → WinDVDClass
- ZoomPlayer → TMainForm

Voilà pour le lecteur multimédia.

3.4.2 Piloter le diaporama intégré à XP

Quoi de plus agréable que de profiter de ses photos de vacances avec sa famille ou ses amis confortablement assis dans un canapé. Windows xp intègre en natif un diaporama simple et bien fait, là encore nous allons utiliser la même méthode de communication, à savoir la simulation de touche clavier. Il nous faut également un type énumerateur avec toutes les commandes que l'on va utiliser (Start, Stop, Suivant, Precedent), rajouter cette déclaration en dessous de celle de TWMPCommand

```
TSlideShowCommand = (swStart, swStop, swNext, swPrev);
```

Les noms sont assez significatif, pas besoin de reprendre leurs désignation.
Le but étant de lancer un diaporama à partir d'un dossier de photo, nous allons utiliser la fonction présente dans la FAQ pour récuperer le chemin vers le dossier Mes images qui se trouve dans le dossier Mes Documents, voilà son implementation

```
{=============================================================================}
{   Renvoie le path d'un dossier spécial de windows                           }
{=============================================================================}
function SpecialFolder(Folder: Integer): String;
var
  SFolder : pItemIDList;
  SpecialPath : Array[0..MAX_PATH] Of Char;
begin
  SHGetSpecialFolderLocation(Form1.Handle, Folder, SFolder);
  SHGetPathFromIDList(SFolder, SpecialPath);
  Result := StrPas(SpecialPath);
end;
```

Ne pas oublier bien sur de rajouter l'unité **Shlobj** dans les uses. Rajouter également l'unité **shellAPI** pour la suite.

La procédure qui va gérer la communication avec le diaporama se présente ainsi

```
{=============================================================================}
{   Commande du diaporama                                                     }
{=============================================================================}
procedure TForm1.RemoteSlideShow(commande: TSlideShowCommand);
var
  ShellExecuteInfo: TShellExecuteInfo;
  ExitCode: DWORD;
  ExecuteFile, ParamString: string;
  hSlideShow: HWND;
begin
  ExecuteFile :='rundll32.exe';
  ParamString := 'shimgvw.dll,ImageView_Fullscreen
'+SpecialFolder(CSIDL_PERSONAL)+'\Mes images';

  FillChar(ShellExecuteInfo, SizeOf(ShellExecuteInfo), 0);
  ShellExecuteInfo.cbSize := SizeOf(TShellExecuteInfo);

  with ShellExecuteInfo do
    begin
      fMask := SEE_MASK_NOCLOSEPROCESS;
      lpFile := PChar(ExecuteFile);
      lpParameters := PChar(ParamString);
      nShow := SW_SHOWNORMAL;
    end;

  if ShellExecuteEx(@ShellExecuteInfo) then                          // on lance notre
process
    begin
      WaitForInputIdle(ShellExecuteInfo.hProcess, INFINITE);         // on attend qu'il
soit initialisé
      hSlideShow := FindWindow('ShImgVw:CPreviewWnd', nil);          // on recherche la
fenêtre du viewer windows
      if hSlideShow <> 0 then
        begin
          setForeGroundWindow(hSlideShow);                           // on la passe au
premier plan
          sleep(500);                                                // on attend 500
ms, le temps qu'elle prenne le focus

          case commande of
            swStart:
              begin
                keybd_event(VK_F11, 0, 0, 0);
                keybd_event(VK_F11, 0, KEYEVENTF_KEYUP, 0);
              end;
            swStop:
              begin
                keybd_event(VK_ESCAPE, 0, 0, 0);
                keybd_event(VK_ESCAPE, 0, KEYEVENTF_KEYUP, 0);
              end;
            swNext:
              begin
                keybd_event(VK_LEFT, 0, 0, 0);
                keybd_event(VK_LEFT, 0, KEYEVENTF_KEYUP, 0);
              end;
            swPrev:
              begin
                keybd_event(VK_LEFT, 0, 0, 0);
                keybd_event(VK_LEFT, 0, KEYEVENTF_KEYUP, 0);
              end;
```

```
          end;
        end;
    end;
end;
```

La diaporama integer à XP n'est ni plus ni moins qu'un dll qui est appelée par l'explorateur. Pour ce faire, nous allons utiliser **rundll32.exe** qui permet d'exécuter une dll comme si c'était une application, celui çi va se charger de lancer la dll avec en paramètre la fonction à exécuter ainsi que les paramètres requis par cette fonction. En observant les points d'entrée de la dll **shimgvw.dll** avec did32.exe (utilitaire de Paul Toth), on aperçoit qu'une fonction **ImageView_Fullscreen** est exportée, elle permet de lancer un diaporama à partir d'un dossier ou une image passé en paramètre, le paramètre de rundll32.exe, devra donc se présenter ainsi

```
rundll32.exe shimgvw.dll,ImageView_Fullscreen pathDuDossierContenantLesImages
```

Pour récupérer le chemin vers le dossier Mes Documents, nous allons utiliser la function **specialFolder()** déclaré un peu avant avec le paramètre **CSIDL_PERSONAL** puis lui rajouter le dossier Mes images.
Ensuite nous allons lancer le processus rundll32.exe en utilisant ShellExecuteEx() car nous aurons besoin de son handle par la suite. Ceci étant fait, après son lancement, il nous faut attendre qu'il soit initialisé et prêt à recevoir les saisies clavier, WaitForInputIdle() nous sera très utile dans ce cas là, cette fonction permet d'attendre l'initialisation d'un process dont le handle est passé en paramètre. Ensuite comme pour le lecteur, on recherche la fenêtre à partir de son className et on la passe au premier plan.
Puis en fonction de la commande, on simule la touche correspondante.

Voilà pour notre projet, vous pouvez télécharger les sources complètes du projet ici.

4 - Conclusion

Ce tutoriel nous as permis d'aborder l'analyse d'un signal, l'utilisation des APIs windows et surtout j'éspère, vous as convaincu q'une bonne recherche d'informations facilite pas mal les choses par la suite et permet de mieux organiser son travail.

5 - Annexes

- Voici une liste des commandes disponibles pour la majorité des télécommandes RC5

Télécommande Philips RC 7531/01			
Touche	Donnée	S5 S4 S3 S2 S1 S0	
TV	Fonction	Binaire	Hexa
0	0	00 0000	0
1	1	00 0001	1
2	2	00 0010	2
3	3	00 0011	3
4	4	00 0100	4
5	5	00 0101	5
6	6	00 0110	6
7	7	00 0111	7
8	8	00 1000	8
9	9	00 1001	9
	Options image	00 1010	A
	Options son	00 1011	B
	Veille	00 1100	C
	Coupure son	00 1101	D
	Volume +	01 0000	10
	Volume –	01 0001	11
	Haut	01 0000	10
	Bas	01 0001	11
	Gauche	01 0101	15
	Droite	01 0110	16
P+	Programme +	10 0000	20
P–	Programme –	10 0001	21
P◂P		10 0010	22
	Menu image	10 1011	2B
	Menu son	10 1100	2C
	Menu options	10 1101	2D
		10 1110	2E
		10 1111	2F
+	Lumière +	11 0010	32
–	Lumière –	11 0011	33
	Teletexte	11 1100	3C

Adresse	A4 A3 A2 A1 A0	
Téléviseur	00000	0
Magnétoscope	00101	5
Récepteur DVB	01010	A

- Une liste des Virtual Keys
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/WinUI/WindowsUserInterface/UserInput/VirtualKeyCodes.asp