Simulazione di un'interfaccia TTY

La volta scorsa abbiamo visto una prima versione di W-LINK, un semplice programma di comunicazione per Windows. Si trattava di una versione funzionante, sia pure limitatamente all'uso di un file di inizializzazione e alla scelta del carattere, mediante la libreria COMMDLG.DLL di Windows 3.1. Passeremo ora al programma completo, aggiungendo l'invio dei caratteri digitati dall'utente e la visualizzazione di quelli ricevuti sullo schermo

di Sergio Polini

Il listato pubblicato nel numero di giugno richiede qualche aggiustamento; oltre all'aggiunta di variabili d'istanza e di metodi alla classe *TCommWindow*, o di alcune costanti, dovremo modificare, in qualche caso, le parti che già conoscete; prenderemo le mosse, quindi, dalla nuova interfaccia della unit COMMWIN (figura 1).

In primo luogo, nella clausola **uses** si aggiunge la unit *Win31*, necessaria per dotare la classe *TCommWin* di un metodo destinato ad intercettare il messaggio WM_COMMNOTIFY. Si aggiungono poi le costanti *cm_Open* e *cm_Close* per le opzioni *Apri* e *Chiudi* del menu *Collegamenti* e, parallelamente, i metodi *CMOpen* e *CMClose*.

Alla classe *TCommWin* si aggiungono, oltre a quelli appena menzionati, i metodi *CanClose* e *CloseComm*, al fine di garantire la chiusura del collegamento non solo quando espressamente richiesta, ma anche quando l'utente esce dall'applicazione; altri metodi, come *SetupWindow*, *Paint* e *ResetScreen* sovraintendono alla visualizzazione del dialogo con l'utente durante un collegamento, compreso l'eventuale cambiamento del carattere usato, mentre *WMSetFocus*, *WMKillFocus* e *MoveCaret* si occupano del cursore. I caratteri digitati dall'utente vengono inviati alla seriale dal metodo *WMChar*; i caratteri ricevuti vengono gestiti dai metodi *WMCommNotify* e *EchoChars*; il metodo *ReportError* avverte di eventuali errori di comunicazione. Nella sezione **private** della classe, compaiono ora la variabile *ComPort* e il metodo *SetCommParams*, destinati la prima a tenere memoria dell'identificativo della porta aperta e il secondo ad impostare i parametri di comunicazione, accanto alle variabili utilizzate per la simulazione di uno schermo di 25 righe per 80 colonne.

```
unit CommWin;
interface
uses
  Win31,WinTypes,WinProcs,Objects,OWindows,ODialogs,Strings,CommDlg;
const
  cm_Open         = 101;
  cm_Close        = 102;
  cm_CommSetup    = 201;
  cm_FontSelect   = 202;
  cm_SaveSetup    = 203;
  cm_RestoreSetup = 204;
  MAXROWS         = 25;
  MAXCOLS         = 80;
type
  PCommWindow = ^TCommWindow;
  TCommWindow = object(TWindow)
    constructor Init(AParent: PWindowsObject; ATitle: PChar);
    procedure SetupWindow; virtual;
    procedure CMOpen(var Msg: TMessage); virtual cm_First + cm_Open;
    procedure CommClose;
    procedure CMClose(var Msg: TMessage); virtual cm_First + cm_Close;
    function CanClose: Boolean; virtual;
    procedure CMCommSetup(var Msg: TMessage);
      virtual cm_First + cm_CommSetup;
    procedure CMFontSelect(var Msg: TMessage);
      virtual cm_First + cm_FontSelect;
    procedure CMSaveSetup(var Msg: TMessage);
      virtual cm_First + cm_SaveSetup;
    procedure CMRestoreSetup(var Msg: TMessage);
      virtual cm_First + cm_RestoreSetup;
    procedure MoveCaret;
    procedure Paint(PaintDC: HDC; var PaintInfo: TPaintStruct); virtual;
    procedure WMSetFocus(var Msg: TMessage);
      virtual wm_First + wm_SetFocus;
    procedure WMKillFocus(var Msg: TMessage);
      virtual wm_First + wm_KillFocus;
    procedure WMChar(var Msg: TMessage); virtual wm_First + wm_Char;
    procedure EchoChars(Chars: PChar; Count: Integer);
    procedure ReportError(ErrorCode: Word);
    procedure WMCommNotify(var Msg: TMessage);
      virtual wm_First + wm_CommNotify;
  private
    Profile: array[0..12] of Char;
    ComPort: Integer;
    LogFont: TLogFont;
    CF: TChooseFont;
    Screen: array[0..MAXROWS-1, 0..MAXCOLS-1] of Char;
    xCursor, yCursor: Integer;
    cxChar, cyChar: Integer;
    procedure ReadProfile;
    procedure WriteProfile;
    function SetCommParams: Boolean;
    procedure ResetScreen;
  end;
```

Figura 1 - La nuova interfaccia della unit COMMWIN.

Nella sezione **implementation** della unit, infine, al termine della dichiarazione delle costanti viene aggiunta una direttiva $I per l'inclusione del file DCBFLAGS.INC, illustrato nel numero di aprile; nella figura 2 sono comunque riprodotte le righe del file necessarie per la compilazione della unit.

```
  fBinary         = $0001;  { modo binario: ignora EofChar              }

  fOutxCtsFlow    = $0008;  { usa CTS per il controllo dell'output      }
  fRtsFlow        = $4000;  { usa RTS per il controllo dell'input       }

  fOutX           = $0100;  { usa XON/XOFF per il controllo dell'output}
  fInX            = $0200;  { usa XON/XOFF per il controllo dell'input }
```

Figura 2 - Le righe del file DCBFLAGS.INC, incluso nella **implementation** *della unit COMMWIN, necessarie per la compilazione della unit.*

Caratteri in finestra

Il dialogo con un computer remoto viene spesso condotto mediante una interfaccia stile TTY: un carattere dopo l'altro, sempre da sinistra verso destra, una riga dopo l'altra, sempre dall'alto verso il basso; proprio come su un normale schermo Unix o MSDOS prima dell'avvento delle interfacce grafiche.

In molti casi è possibile ottenere effetti più interessanti, come finestre, colore e movimenti del cursore, mediante le sequenze di escape ANSI; il programma in esecuzione sul computer locale, quando riceve un carattere Esc seguito da una parentesi quadra sinistra, interpreta i caratteri successivi come un comando: un «RF, ad esempio, esige la cancellazione della riga corrente a partire dalla posizione del cursore, un «2J» vuole la cancellazione di tutto lo schermo, ecc. Si tratta, nel mondo MSDOS, di quei comandi che vengono attivati caricando ANSI.SYS.

Il programma W-Link potrebbe essere modificato in modo da tener conto delle sequenze ANSI, ma ne risulterebbe un listato sensibilmente più lungo di quanto non sia già ora; si tratterebbe, inoltre, di un esercizio di «normale programmazione», mentre, in questa sede, ci interessano soprattutto tecniche proprie della programmazione per eventi. L'interfaccia, quindi, rimane fedele allo stile TTY: si simula uno schermo di 25 righe per 80 colonne, con il tradizionale trattino lampeggiante come cursore. Si usano, a questo scopo, un array *Screen* di 25x80 caratteri e due variabili *xCursor* e *yCursor* per la posizione del cursore, l'uno e le altre inizializzati dal constructor *TCommWindow.Init* (figura 3). Ci si avvale, infine, del *mapping mode* MM_TEXT che, comunque, non è necessario impostare esplicitamente, in quanto attivo per default (ho accennato ai *mapping mode* nell'articolo *Programmare sotto Windows*, compreso nel supplemento che il numero di aprile di MC ha dedicato a Windows). In una tale interfaccia sarebbe lecito attendersi un carattere come *Terminal* (così chiamato per gli utenti; per i programmatori

```
constructor TCommWindow.Init(AParent: PWindowsObject; ATitle: PChar);
begin
  TWindow.Init(AParent, ATitle);
  Attr.Menu := LoadMenu(HInstance, 'WLINKMENU');
  EnableMenuItem(Attr.Menu, cm_Close,
                 mf_ByCommand + mf_Grayed + mf_Disabled);
  StrLCopy(Profile, ATitle, 8);
  Profile[7] := #0;
  StrCat(Profile, '.INI');
  ReadProfile;
  ComPort := -1;
  FillChar(Screen, MAXROWS*MAXCOLS, ' ');
  xCursor := 0;
  yCursor := 0;
end;

procedure TCommWindow.SetupWindow;
begin
  inherited SetupWindow;
  GetObject(GetStockObject(OEM_Fixed_Font), SizeOf(LogFont), @LogFont);
  ResetScreen;
end;

procedure TCommWindow.ResetScreen;
var
  DC: HDC;
  Font: HFont;
  TM: TTextMetric;
  X, Y, Width, Height: Integer;
begin
  DC := GetDC(HWindow);
  Font := SelectObject(DC, CreateFontIndirect(LogFont));
  GetTextMetrics(DC, TM);
  DeleteObject(SelectObject(DC, Font));
  ReleaseDC(Hwindow, DC);
  cxChar := TM.tmAveCharWidth;
  cyChar := TM.tmHeight + TM.tmExternalLeading;
  Width  := cxChar * MAXCOLS + GetSystemMetrics(SM_CXFRAME) * 2;
  Height := cyChar * MAXROWS + cyChar div 2 +
            GetSystemMetrics(SM_CYFRAME) * 2 +
            GetSystemMetrics(SM_CYCAPTION) +
            GetSystemMetrics(SM_CYMENU);
  X := (GetSystemMetrics(SM_CXSCREEN) - Width) div 3;
  Y := (GetSystemMetrics(SM_CYSCREEN) - Height) div 3;
  MoveWindow(HWindow, X, Y, Width, Height, True);
end;
procedure TCommWindow.CMFontSelect(var Msg: TMessage);
begin
  FillChar(CF, SizeOf(CF), 0);
  CF.lStructSize := SizeOf(CF);
  CF.hwndOwner := HWindow;
  CF.lpLogFont := @LogFont;
  CF.Flags := CF_FIXEDPITCHONLY or CF_SCREENFONTS or CF_EFFECTS
              or CF_INITTOLOGFONTSTRUCT;
  CF.nFontType := SCREEN_FONTTYPE;
  if ChooseFont(CF) then begin
    HideCaret(HWindow);
    ResetScreen;
    ShowCaret(HWindow);
    MoveCaret;
  end;
end;
```

Figura 3 - I metodi della classe TCommWindow *che provvedono* a preparare l'output sullo schermo.

```
procedure TCommWindow.WMSetFocus(var Msg: TMessage);
begin
  CreateCaret(HWindow, 0, cxChar, 0);
  ShowCaret(HWindow);
  MoveCaret;
end;

procedure TCommWindow.WMKillFocus(var Msg: TMessage);
begin
  HideCaret(HWindow);
  DestroyCaret;
end;
```

Figura 4 - I metodi che creano e visualizzano, e poi nascondono e distruggono il caret, *ogni volta che l'applicazione diventa o smette di essere quella con cui interagisce l'utente.*

```
procedure TCommWindow.MoveCaret;
var
  DC: HDC;
  Font: HFont;
  x: Integer;
begin
  if HWindow = GetFocus then
    if ComPort >= 0 then
      SetCaretPos(xCursor * cxChar, (yCursor + 1) * cyChar)
    else
      SetCaretPos(-1, -1);
end;
```

Figura 5 - Il metodo che, muovendo il caret, *tiene aggiornata la posizione del cursore.*

si chiama OEM_FIXED_FONT); già sapete, tuttavia, che in W-Link l'utente può scegliere un carattere diverso. I motivi sono due; in primo luogo, una finestra che consenta di vedere 25 righe di 80 caratteri *Terminal* occupa quasi tutto un video VGA; in secondo luogo, questo mi è sembrato un buon motivo per estendere il tema della nostra chiacchierata fino a ricomprendervi anche la scelta dei vari tipi di carattere mediante la libreria COMMDLG.DLL. Ciò comporta, però, che la finestra di W-Link deve sapersi adeguare alla dimensione del carattere: ho verificato che è piuttosto noioso, una volta scelto un carattere, cambiare le dimensioni della finestra in modo che vi entri tutto quello che vi deve entrare. Perché non lasciare che ci pensi il programma?

Nel metodo *SetupWindow*, una volta chiamato l'omonimo metodo della classe madre (che sarebbe *TWindow*; la clausola **inherited** del Pascal 7.0 consente di lasciare al compilatore la ricerca della classe immediatamente precedente nella gerarchia), si inizializza la variabile *Logfont* con i valori di un carattere predefinito (di *stock*), OEM_FIXED_FONT; si chiama quindi *ResetScreen* per preparare l'outupt su una finestra con le giuste dimensioni.

La variabile *LogFont* viene inizializzata per semplificare la gestione economica delle risorse di sistema. Quando si seleziona un carattere per un *device context* con *SelectObject*, questo occupa spazio nel segmento dati di GDI.EXE, spazio che andrebbe rilasciato non appena possibile con *DeleteObject*. La funzione *DeleteObject* non può però essere usata con i caratteri predefiniti. Dal momento che l'utente può scegliere tra diversi tipi di carattere, sarebbe un po' acrobatico pretendere di distinguere tra tipi predefiniti o meno; è preferibile, quindi, lavorare sempre con tipi di carattere *logici*, anche se questo comporta iniziare con un tipo logico del tutto equivalente al tipo di *stock*. Aggiungo che sarebbe possibile integrare i metodi *WriteProfile* e *ReadProfile*, visti il mese scorso, in modo da consentire all'utente di salvare anche il carattere scelto (i valori dei campi di *LogFont*) ed iniziare con questo una successiva sessione; ciò non avviene in W-Link unicamente per evitare di allungare troppo il listato. Diciamo che è un facile esercizio per il lettore...

Il metodo *ResetScreen* seleziona temporaneamente un carattere creato sulla base dei valori dei campi di *LogFont*, al fine, in primo luogo, di avvalorare le variabili *cxChar* e *cyChar* con le dimensioni del carattere: l'ampiezza media (*tmAveCharWidth*) e l'altezza (*tmHeight*), quest'ultima incrementata con quanto necessario per una ragionevole interlinea (*tmExternalLeading*). Le misure sono espresse in unità logiche, ma questo, nel nostro caso, non è un problema; siamo infatti nel modo MM_TEXT, nel quale unità logiche e fisiche (i pixel) coincidono. Si può quindi procedere senza ulteriori formalità nella determinazione della posizione e della dimensione che la finestra dovrà avere. Le dimensioni si ottengono aggiungendo allo spazio necessario per i caratteri (*cxChar* per 80 e *cyChar* per 25) quello occupato dalla cornice della finestra, dal suo titolo e dalla barra del menu, ottenuto chiamando la funzione *GetSystemMetrics* (che rende le misure in pixel). La posizione viene determinata sulla base delle dimensioni dello schermo (sempre ottenute con *GetSystemMetrics*). La funzione *MoveWindow* si incarica di ridisegnare la finestra, in quanto chiamata con TRUE come ultimo parametro.

ResetScreen è anche chiamato, ovviamente, dal metodo *CMFontSelect* nella sua versione definitiva, illustrata anch'essa, come i metodi precedenti, nella figura 3. Da notare che, rispetto alla versione proposta il mese scorso, ora non vi è solo qualcosa in più, ma anche qualcosa di diverso. Cambia, infatti, l'avvaloramento del campo *CF.Flags*: scompare la costante CF_TTYONLY, sostituita da CF_FIXEDPITCHONLY, e si aggiunge CF_INITTOLOGFONTSTRUCT. Quest'ultima costante fa sì che, quando la funzione *ChooseFont* apre la dialog box, in essa appaia il nome del carattere descritto nei valori correnti dei campi di *LogFont*; l'altra limita la scelta ai soli caratteri non proporzionali, sicuramente più adeguati ad un'interfaccia stile TTY (sarebbe possibile usare caratteri proporzionali; ne risulterebbero però non solo un output, per così dire, innaturale, ma anche la necessità di complicare – o cambiare radicalmente – i relativi metodi; nelle situazioni in cui si giunge all'ultima riga e occorre, quindi, uno scorrimento verticale, con il codice attuale e un carattere proporzionale le righe «corte» verrebbero «sporcate» dai residui di righe precedenti più lunghe; l'ampiezza dei singoli caratteri, infatti, è sempre uguale a quella media nei caratteri non proporzionali, ma può essere ben diversa in quelli proporzionali. Occorrerebbe almeno riscrivere la procedura *DoLineFeed* dichiarata nel metodo *EchoChars*, che

vedremo in seguito, ad esempio usando la funzione *ScrollWindow*).

Se l'utente chiude la dialog box per la scelta del carattere premendo il tasto «OK», la funzione *ChooseFont* ritorna TRUE; in tal caso, si nasconde il cursore, si cambiano posizione e dimensioni della finestra, si rende nuovamente visibile il cursore e se ne aggiorna la posizione.

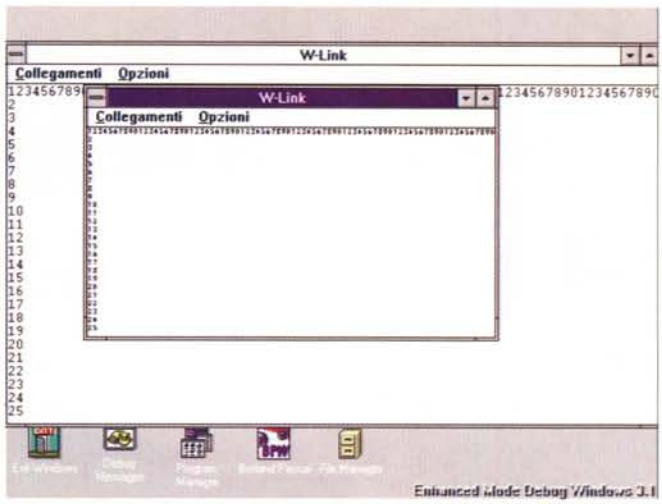

Figura 6 - Due istanze del programma W-Link, con due tipi di caratteri diversi e, conseguentemente, finestre di diverse dimensioni.

Il caret come cursore

Sotto MSDOS per cursore si intende quel familiare trattino lampeggiante che indica costantemente dove apparirà il carattere che ci apprestiamo a digitare sulla tastiera; in Windows per cursore si intende, invece, l'immagine (freccia, croce, clessidra, ecc.) che indica la posizione corrente del mouse; al cursore tipico di un'interfaccia non grafica corrisponde il *caret,* spesso rappresentato come una linea verticale.

Un'applicazione può usare il caret che desidera: una linea o un rettangolo orizzontale o verticale, o anche un'immagine qualsiasi memorizzata in una bitmap; sarà in ogni caso un'entità lampeggiante, che potrà essere mossa a piacimento nella finestra che la ospita. Nel nostro caso, ovviamente, sceglieremo un caret-cursore (nel senso di MSDOS), un trattino largo quanto un carattere e alto quanto è spesso il bordo di una finestra. La prima misura altro non è che *cxChar;* quanto alla seconda, potremmo usare la funzione *GetSystemMetrics* passandole come parametro la costante SM_CYBORDER, ma è sufficiente usare uno zero come quarto parametro della funzione CreateCaret (per bordo di una finestra si intende la linea che delimita le finestre le cui dimensioni non possono essere modificate; per cornice – *frame* – si intende quella linea «spessa» che delimita le finestre le cui dimensioni possono essere modificate «trascinando» con il mouse lati e vertici della cornice; si usa lo spessore del bordo per essere sicuri di creare un cursore sottile ma sicuramente visibile quale che sia la risoluzione del video).

Il caret è una risorsa condivisa da tutte le applicazioni; ciò comporta che ogni applicazione deve creare il proprio caret quando intrattiene il dialogo con l'utente (quando ha il *focus*) e distruggerlo ogni volta che l'utente si sposta su altre applicazioni. Ciò si ottiene agevolmente intercettando i messaggi WM_SETFOCUS e WM_KILLFOCUS, come indicato nella figura 4.

Il caret può essere o meno visibile: lo si rende visibile con *ShowCaret,* lo si nasconte con *HideCaret*. Normalmente, il caret si vede dopo che è stato creato, non si vede dopo che è stato distrutto; in altri termini, il caret si vede sempre quando l'applicazione ha il *focus*. Nel caso di W-Link, tuttavia, il caret segue anche un'altra regola: non si vede se non dopo che si è aperta una porta seriale e solo fintanto che questa rimane aperta. Ricorderete che, chiamando la funzione *OpenComm*, questa ritorna un intero che può essere maggiore o uguale a zero in caso di successo, negativo in caso di errore; se tutto va bene, quell'intero nullo o positivo va usato come identificativo della porta aperta nelle altre funzioni della API di Windows preposte al controllo della comunicazione seriale. La classe *TCommWindow* ha quindi una variabile privata *ComPort,* che viene inizializzata a -1 (cfr. il constructor nella figura 3) ad indicare lo stato di «porta non aperta»; vedremo che il metodo *CommClose* assegna nuovamente -1 a *ComPort* quando si chiude il collegamento. Il valore di *ComPort,* quindi, può essere usato per determinare se il caret deve risultare visibile o meno, e ciò avviene nel metodo *MoveCaret* (figura 5), che si fa carico di tenere aggiornata la posizione sullo schermo del nostro caret-cursore.

MoveCaret e *CommClose* potrebbero usare il metodo *HideCaret* per rendere invisibile il caret, così come si potrebbe usare *ShowCaret* una volta che la porta risultasse aperta con successo in *CMOpen,* ma ciò non risulterebbe semplice. I metodi che mostrano e nascondono il caret, infatti, funzionano in modo «relativo», non «assoluto»: perché ShowCaret mostri il caret, ad esempio, non è necessario solo che questo sia stato creato dall'applicazione la cui finestra ha il *focus,* ma anche che non sia stato nascosto due o più volte; dopo due o più chiamate di *HideCaret* occorrono almeno altrettante chiamate di *ShowCaret* per poter rivedere il caret, e viceversa. Ne segue che, se si mostrasse e si nascondesse il caret non solo secondo il *focus,* ma anche secondo lo stato della porta, bisognerebbe tenere un conteggio accurato delle diverse circostanze in cui si agisce sulla sua visibilità. Basterebbe magari una variabile-contatore, ma è forse più semplice la soluzione adottata in W-Link: quando si inizia la porta non è aperta, quindi *ComPort* vale -1, quindi *MoveCaret* sposta il caret in una posizione (-1, -1) esterna ai limiti della finestra, rendendolo così invisibile (vedremo che *CommClose* opera in modo analogo); una volta aperta la porta, il caret viene posizionato secondo il numero di caratteri della riga corrente (la variabile *xCursor,* aggiornata dal metodo *EchoChars*) e l'ampiezza dei caratteri (*cxChar*); il tutto, naturalmente, solo nel caso la finestra di W-Link risulti avere il *focus*.

Per ora ci fermiamo qui. Vedremo a settembre, dopo un'estate che auguro a tutti serena, come imbrigliare nell'interfaccia appena descritta il dialogo con il computer remoto. MC

Sergio Polini è raggiungibile tramite MC-link alla casella MC1166 e tramite Internet all'indirizzo MC1166@mclink.it.