La ricezione di caratteri come «evento»

La volta scorsa abbiamo iniziato l'esame delle funzioni della API di Windows preposte all'uso delle porte seriali. Ci siamo soffermati sulle funzioni OpenComm *e* SetCommState, *mediante le quali si apre una porta e la si imposta secondo i parametri richiesti per la comunicazione con un modem (e un computer) remoto. Vedremo ora le funzioni per la trasmissione e ricezione di caratteri, ma soprattutto come gestire la ricezione di caratteri nell'ambito della programmazione per eventi*

di Sergio Polini

La API di Windows comprende una quindicina di funzioni per la gestione delle porte seriali; non sono tutte necessarie in ogni occasione, alcune sono anche un po' ridondanti. *BuildCommDCB*, ad esempio, vuole come parametri una stringa e una variabile di tipo *TDCB* (il record che abbiamo esaminato il mese scorso); la stringa deve avere lo stesso formato dei parametri con cui si userebbe il comando del DOS MODE, ad esempio "com1:96,n,8,1", e viene usata per avvalorare i campi del record *TDCB* relativi a nome della porta, velocità di trasmissione, parità, bit di dati, bit di stop, ma non anche quelli relativi all'*handshake* hardware (RTS/CTS) o software (XON/XOFF); per rendere effettive le impostazioni, inoltre, va comunque usata la funzione *SetCommState*. La funzione *GetCommState* può essere usata per leggere l'impostazione corrente di una porta.

La funzione *SetCommBreak* sospende la trasmissione di caratteri e pone la porta in uno stato di «break» fino alla chiamata della funzione *ClearCommBreak*. *TransmitCommChar* pone un carattere all'inizio della coda di output, provocandone la trasmissione immediata, con precedenza rispetto ad altri caratteri in coda. *FlushComm* svuota le code di input o di output, secondo il valore del suo secondo parametro (zero per la coda di trasmissione, uno per quella di ricezione). *UngetCommChar* pone un carattere all'inizio della coda di ricezione, in modo che venga letto per primo alla successiva lettura (non sono consentite chiamate consecutive della funzione; il carattere posto nella coda deve essere letto prima che si possa ripetere l'operazione). *CloseComm* chiude la porta, dopo aver trasmesso tutti i caratteri eventualmente presenti nella coda di output, e rilascia la memoria allocata per le due code.

Con *EscapeCommFunction* è possibile agire sullo stato della porta, precisando nel secondo parametro il tipo di azione che si intende portare a termine, mediante le costanti riportate nella figura 1.

Trasmissione di caratteri

Dopo aver aperto la porta con *OpenComm* e impostato i parametri di comunicazione con *SetCommState*, si potrebbe «quasi» procedere solo con le funzioni *ReadComm* e *WriteComm*, per ricevere e inviare caratteri, concludendo poi con *CloseComm*. Un primo motivo per quel «quasi» è la possibilità di errori.

Immaginiamo, naturalmente, di dover scrivere un programma che gestisca una comunicazione con un computer remoto, ad esempio il computer che, in redazione, ospita MC-link (figura 2). A parte l'invio di file, verranno trasmessi i caratteri digitati sulla tastiera; in un'applicazione Windows, ciò viene realizzato in modo molto semplice, in quanto si tratta solo di intervenire in occasione dei messaggi WM_CHAR.

La funzione *WriteComm*, tuttavia, potrebbe non riuscire ad inviare un carattere; può succedere, ad esempio, che la coda di output sia piena. *WriteComm* vuole come parametri un intero che identifichi la porta (quello fornito da *OpenComm*), una stringa e un intero che indichi il numero dei caratteri da trasmettere; il suo risultato sarà il numero dei caratteri effettivamente trasmessi, positivo se tutto è andato bene, negativo se è intervenuto qualche errore.

Nel caso più semplice, si passerà come stringa da trasmettere una stringa contenente un solo carattere, quello il cui codice ASCII sarà contenuto nel campo *WParamLo* del messaggio WM_CHAR e, quindi, un errore provocherà un risultato minore di (o comunque diverso da) 1.

L'eventuale errore va subito riconosciuto, in quanto, in caso di errore, Windows blocca la porta fino alla chiamata della funzione *GetCommError*. Questa richiede un parametro variabile di tipo *TComStat* (figura 3), attraverso il quale è possibile ottenere informazioni supplementari. Anche qui, però, ci si imbatte in un problema già visto il mese scorso a proposito del record *TDCB*: una versione in Pascal di una struttura in C contenente diversi campi di tipo *bitfield*, raggruppati in un unico campo *Flags* per l'accesso al quale non si offre alcun aiuto. Vi propongo, quindi, le costante della figura 4, attraverso le quali sarà facile verificare, con un'operazione **and**, se un dato bit è settato o meno in quel campo *Flags*. Gli altri campi di *TComStat* sono destinati ad ospitare il numero dei caratteri presenti, rispettivamente, nella coda di input e in quella di output.

CLRDTR	abbatte il segnale DTR
CLRRTS	abbatte il segnale RTS
RESETDEV	resetta la porta
SETDTR	invia il segnale DTR
SETRTS	invia il segnale RTS
SETXOFF	simula la ricezione del carattere XOFF
SETXON	simula la ricezione del carattere XON

Figura 1 - Le costanti mediante le quali si può specificare il tipo di azione da portare a termine con la funzione EscapeCommFunction. *I diversi segnali sono stati illustrati il mese scorso (figura 6).*

Per ottenere informazioni sull'errore occorso, tuttavia, e per riattivare la comunicazione, è sufficiente esaminare il risultato di *GetCommError*, che può essere una combinazione di più costanti. I nomi e i valori di queste sono definiti nella unit WINTYPES; le ripropongo comunque nella figura 5, anche per raccoglierle in gruppi omogenei e per isolare quelle — le ultime — relative alla stampante e alla porta parallela. Se interessa solo il risultato della funzione si può usare una variabile «nulla» come secondo argomento. Questo è facile in C, in quanto il secondo parametro è un puntatore e può essere sostituito con la costante NULL (zero); in Pascal però (nonostante quello che dice la documentazione Borland sulla funzione...) non si può usare **nil** per un parametro variabile. Si può aggirare il problema con un trucchetto come quello cui ho fatto ricorso nella figura 6, che illustra l'uso «tipico» della funzione *WriteComm* nell'ambito di un metodo che intercetti il messaggio WM_CHAR: una variabile locale dichiarata con la clausola **absolute** come residente all'indirizzo 0:0.

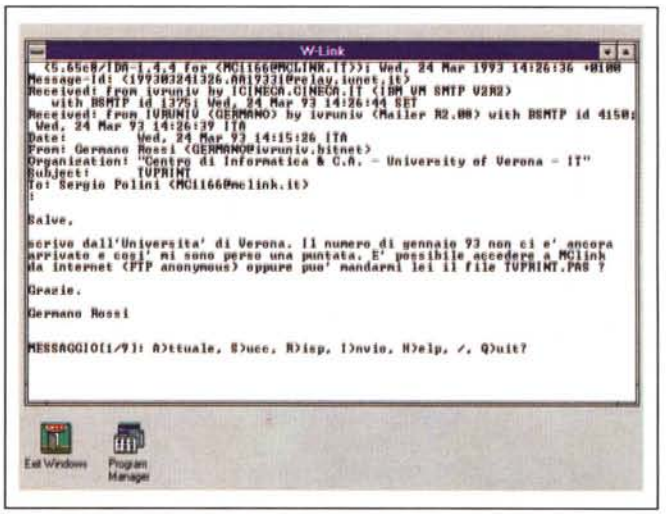

Figura 2 - Un semplice programma di comunicazione all'opera: dopo il collegamento con MC-link, leggo un messaggio inviatomi via Internet da Germano Rossi, dell'Università di Verona.

Il Message Loop

Trasmettere un carattere, in definitiva, è piuttosto semplice; riceverne uno è un'altra storia.

I messaggi sono il motore di un'applicazione Windows; per guidare bene questo motore, occorre evitare di interferire con un meccanismo che prevede risposte rapide a messaggi che giungono in rapida successione; la risposta ad un messaggio deve limitarsi a fare quanto strettamente necessario per trattarlo, restituendo poi il controllo a Windows per consentirgli di continuare a smistare messaggi.

In ObjectWindows, questo vuol dire prevedere un metodo virtuale per ogni messaggio che si intenda trattare; vi sono però situazioni che non possono essere gestite in questo modo, in quanto non sono fonti di messaggi. In Windows 3.0 la comunicazione seriale era una di queste situazioni; era quindi necessario ricorrere ad altri strumenti.

```
TComStat = record
  Flags: Byte;
  cbInQue: Word;
  cbOutQue: Word;
end;
```

Figura 3 - Il record TComStat, *usato per ottenere informazioni sullo stato di una porta mediante la funzione* GetCommError.

Per amore di completezza, diremo brevemente che sarebbe possibile installare un timer, in modo da controllare periodicamente la presenza di caratteri nella coda di ricezione. Ma si tratterebbe di un approccio poco elegante e non privo di inconvenienti (non ultimi il numero limitato e la scarsa accuratezza dei timer di Windows).

Il metodo più corretto, nonostante quanto dicano i manuali Borland dei Turbo Pascal per Windows 1.0 e 1.5, che sconsigliano di ridefinire il metodo *TApplication.MessageLoop* (figura 7), ri-

```
const
  fCtsHold  = $01;     { trasmissione sospesa in attesa di un CTS }
  fDsrHold  = $02;     { trasmissione sospesa in attesa di un DSR }
  fRlsdHold = $04;     { trasmissione sospesa in attesa di un RLSD }
  fXoffHold = $08;     { trasmissione sospesa a seguito della
                         ricezione di uno XOFF }
  fXoffSent = $10;     { trasmissione sospesa a seguito della
                         trasmissione di uno XOFF }
  fEof      = $20;     { ricevuto carattere di fine input }
  fTxim     = $40;     { c'e' un carattere in attesa di essere
                         trasmesso }
```

Figura 4 - Le costanti che possono essere usate per leggere il campo Flags *di un record* TComStat.

```
CE_BREAK    = $0010;   { riscontrata una condizione di break }

CE_FRAME    = $0008;   { bit di stop non ricevuto al momento giusto }
CE_OVERRUN  = $0002;   { un nuovo carattere e' arrivato prima che il
                         precedente sia stato letto }
CE_RXPARITY = $0004;   { controllo di parita' con esito negativo }

CE_CTSTO    = $0020;   { timeout nella ricezione di CTS }
CE_DSRTO    = $0040;   { timeout nella ricezione di DSR }
CE_RSLDTO   = $0080;   { timeout nella ricezione di RLSD }

CE_RXOVER   = $0001;   { overflow nella coda di ricezione }
CE_TXFULL   = $0100;   { overflow nella coda di trasmissione }

CE_MODE     = $8000;   { richiesta non supportata o identificativo
                         della porta non valido }

CE_PTO      = $0200;   { timeout sulla porta parallela       }
CE_IOE      = $0400;   { errore di I/O sulla porta parallela }
CE_DNS      = $0800;   { porta parallela non selezionata     }
CE_OOP      = $1000;   { stampante senza carta               }
```

Figura 5 - Le costanti usate per il risultato della funzione GetCommError.

chiede una ridefinizione di questo. È necessario, infatti, intervenire nel loop che verifica la presenza di messaggi e, in caso positivo, li mette a disposizione della finestra cui sono destinati; è necessario sostituire la funzione *GetMessage*, che ritorna FALSE solo quando intercetta il messaggio WM_QUIT, con *PeekMessage*, che ritorna FALSE quando non vi sono messaggi in attesa, al fine di poter «fare altro» in queste circostanze. Occorre un loop basato su *PeekMessage* che, se vi sono messaggi esca nel caso il messaggio sia WM_QUIT e tratti altri messaggi come nella figura 7, se non vi sono messaggi dia modo di «fare altro».

```
TClasse = object(TWindow)
  ...
  procedure ReportError(ErrorCode: Integer);
  procedure WMChar(var Msg: TMessage); virtual wm_First + wm_Char;
  ...
private
  ComPort: Integer;
  ...
end;
...
procedure TClasse.WMChar(var Msg: TMessage);
var
  NULL: TComStat absolute 0000:0000;
begin
  if WriteComm(ComPort, @Msg.WParamLo, 1) <> 1 then
    ReportError(GetCommError(ComPort, NULL));
end;
```

Figura 6 - Implementazione del metodo WMChar *per una generica classe* TClasse *con la quale si voglia inviare caratteri tramite la porta seriale (*ComPort *avrà come valore il risultato di* OpenComm*).*

```
procedure TApplication.MessageLoop;
var
  Message: TMsg;
begin
  while GetMessage(Message, 0, 0, 0) do begin
    if not ProcessAppMsg(Message) then begin
      TranslateMessage(Message);
      DispatchMessage(Message);
    end;
  end;
  Status := Message.WParam;
end;
```

Figura 7 - Il sorgente del metodo TApplication.MessageLoop *come era nelle versioni 1.0 e 1.5 del Turbo Pascal per Windows. Si esce dal ciclo* **while** *solo quando si riceve il messaggio WM_QUIT.*

Ciò è confermato dal metodo *TApplication.MessageLoop* come implementato prima nel Borland C++ 3.1, poi nel Borland Pascal 7.0: viene usato un loop nel cui ambito si chiama un metodo *IdleAction* che è possibile ridefinire ai propri scopi. Si tratta proprio di uno di quegli «agganci» cui mi riferivo il mese scorso, di quei punti dei meccanismi della programmazione per eventi in cui si deve poter intervenire per ottenere l'effetto giusto nel momento giusto (tanto che, se non ci fosse, dovremmo ridefinire *MessageLoop* secondo la pseudocodifica della figura 8). Nel nostro caso, potremo usare *IdleAction* per verificare la presenza di caratteri nella coda di ricezione e, in caso affermativo, prelevarli.

A questo scopo, potremmo usare la funzione *GetCommState* e, se il campo *cbInQue* avesse un valore maggiore di zero, usare *ReadComm* per leggere i caratteri; potremmo però anche usare solo *ReadComm* e, in caso di risultato nullo o negativo, chiamare *GetCom-*

L'angolo di Internet

Approfittando della connessione tra MC-link e Internet, Cristiano, studente di Ingegneria Elettronica presso il Politecnico di Milano, mi chiede: «Sto scrivendo un programma Pascal e ho la necessità di stampare in un *TRect* del testo. Il problema è che se il testo è più largo del rettangolo, usando la funzione API *ExtTextOut* con opzione ETO_Clipped il testo viene, ovviamente, troncato. Esiste la possibilità di giustificare automaticamente la stringa oppure devo farlo a mano? Come si usano o come si potrebbero usare le funzioni *SetTextJustification* e simili?».

Gli rispondo volentieri su queste pagine, in quanto si tratta di un problema che, generalizzato, potrebbe riguardare anche il programma di comunicazione che vedremo il mese prossimo: come scrivere testo (ad esempio, il dialogo con un sistema come MC-link) in un rettangolo (la finestra mediante la quale avviene il dialogo) che potrebbe avere dimensioni variabili (come ogni finestra Windows che si rispetti)?

Per «riempire» il rettangolo senza lasciare troppo spazio vuoto e senza problemi di debordi (cioè di clipping), si può scegliere un font *true type* e cambiarne le dimensioni in modo da adattarle a quelle del rettangono, secondo un criterio che sia più o meno: BaseRettangolo x *AltezzaRettangolo = NumeroCaratteri x tm.tmAveCharWidth x (tm.tmHeight + tm.tmExternalLeading)*, dove *tm* è una variabile di tipo *TTextMetric*; si deve poi impedire all'utente di restringere il rettangolo oltre un limite dato dalla dimensione minima disponibile per il font.

Il testo può poi essere disposto nel rettangolo con word-wrapping e giustificazione. Per il word-wrapping basta lavorare sugli spazi bianchi (a meno che non si voglia anche la divisione automatica in sillabe); per la giustificazione si può effettivamente usare la funzione *SetTextJustification*. Ambedue i problemi sono trattati con chiarezza nel libro di Petzold, in un paragrafo intitolato «On-line Text Alignment» (nel cap. 14, «Text and Fonts»). In breve:

Siano *xRight* e *xLeft* i margini destro e sinistro del rettangolo. Una volta costruita una riga *Text* che non sia né troppo lunga né troppo corta (tale cioè che aggiungendo una sola altra parola si andrebbe fuori margini), se ne calcola la lunghezza con:

```
Extent := GetTextExtent(hDC, Text, strlen(Text));
```

In pratica, si tratta di costruire un ciclo in cui, partendo da una data posizione assunta come inizio della riga, si contano gli spazi e si calcola *Extent* fino a che si va oltre *xRight-xLeft*; a questo punto, la riga terminerà con lo spazio individuato subito prima di ottenere un *Extent > xRight-xLeft*.

Si chiama quindi:

```
SetTextJustification(hDC, xRight-xLeft-LOWORD(Extent), nBlank);
```

dove *nBlank* è il numero degli spazi presenti in *Text* (e quindi delle zone da «allargare»).

Subito dopo, per stampare la riga giustificata:

```
TextOut(hDC, xLeft, y, Text, strlen(Text));
```

Bisogna solo stare attenti a chiamare *SetTextJustification(hDC, 0, 0)* prima di cominciare una nuova riga.

SetTextJustification, infatti, accumula internamente «errori» se lo spazio da aggiungere per giustificare non può essere distribuito equamente tra gli *nBlank* spazi, e *GetTextExtent* tiene conto di questi «errori». Quando si inizia una nuova riga, quindi, bisogna cancellare quegli «errori» chiamando *SetTextJustification* con parametri nulli.

Dal mio punto di vista, questa breve digressione dimostra che, pure a costo di dover imparare un po' di C, il libro di Petzold è obbligatorio anche per chi usa altri linguaggi.

```
TApplication.MessageLoop
var
  Message: TMsg
begin
  inizio_ciclo
    se PeekMessage(Message, 0, 0, 0, PM_REMOVE) = VERO
      se Message.Message = WM_QUIT
        esci
      se ProcessAppMsg(Message) = FALSO
        TranslateMessage(Message);
        DispatchMessage(Message);
    altrimenti
      (* se PeekMessage e' FALSO e, quindi, non vi sono messaggi *)
      fai_altro
  fine_ciclo
  Status := Message.WParam
end
```

Figura 8 - Pseudocodifica di una diversa implementazione del metodo TApplication.MessageLoop. *La troviamo nel Borland C++ 3.1 e nel Borland Pascal 7.0, che dispongono di un metodo* IdleAction *per «fai_altro».*

mError analogamente a quanto abbiamo già visto per la trasmissione (figura 9). Va ricordato che, quando il risultato è negativo, esprime comunque il numero dei caratteri effettivamente letti — quindi prelevati dalla coda di ricezione — prima dell'errore.

Notificazione di eventi

L'intervento diretto o indiretto sul *message loop* non rappresenta comunque una soluzione perfetta; si impone un'alternativa, infatti, tra il meccanismo normale, basato su eventi e messaggi, e una sorta di polling intermittente sulla porta. Si rischia o di intervenire sulla porta anche quando non c'è alcun carattere, o di ritardare l'intervento a causa di un intenso traffico di messaggi.

Con la API di Windows 3.1 si dispone finalmente di una soluzione migliore. Erano già presenti nella versione 3.0 le funzioni *SetCommEventMask* e *GetCommEventMask*. Alla porta è associata una *word* i cui bit possono essere settati per indicare il verificarsi di un evento; *SetCommEventMask* consente di abilitare alcuni di questi eventi mediante il parametro *EvtMask*, che deve essere una combinazione delle potenze di 2 espresse mediante le costanti riportate nella figura 10 (che comprende anche le costanti con valore superiore a $0200, aggiunte nella versione 3.1); il risultato è una combinazione degli stessi valori che indica quali eventi si sono verificati. Per venire a conoscenza degli eventi che si sono verificati, si dispone della funzione *GetCommEventMask*, il cui risultato va interpretato come quello di *SetCommEventMask*; il parametro *EvtMask* permette di precisare quali eventi vanno azzerati dopo la lettura.

Il problema, fino alla versione 3.0, era uno solo: quando leggere gli eventi? Nel *message loop*, col rischio di appesantirlo ulteriormente? Con Windows 3.1 si dispone finalmente di una funzione *EnableCommNotification*, mediante la quale si abilita l'invio ad una data finestra (il secondo parametro) di un messaggio WM_COMMNOTIFY generato da una porta (il primo parametro) che sia stata abilitata a segnalare eventi mediante la funzione *SetCommEventMask*. Quanto si riceve un messaggio WM_COMMNOTIFY, si deve esaminare *LParamLo*, che può essere una combinazione di uno o più dei flag CN_EVENT, CN_RECEIVE o CN_TRANSMIT; nel caso di CN_EVENT, va usata *GetCommEventMask* per riconoscere l'evento e azzerarlo.

La vera novità è però rappresentata dalla possibilità di ricevere il messaggio in occasione della ricezione di caratteri. Il terzo e il quarto parametro, *cbWriteNotify* e *cbOutQueue*, indicano rispettivamente quanti caratteri devono trovarsi nella coda di input prima che venga inviato il messaggio, e il numero di caratteri nella coda di output al di sotto del quale viene segnalata col messaggio l'opportunità di scriverne altri. Se si usa –1 per *cbWriteNotify*, non viene usato il flag CN_RECEIVE; se si usa –1 per *cbOutQueue*, non viene usato il flag CN_TRANSMIT.

In concreto, per applicazioni che vogliano implementare la comunicazione con un computer remoto come l'host di MC-link, si tratta di chiamare la funzione con –1 come quarto parametro e di predisporre un metodo per il messaggio WM_COMMNOTIFY.

Vedremo il mese prossimo come procedere per mettere all'opera il tutto in un semplice programma di comunicazione. MC

Figura 9 - Esempio di un metodo IdleAction *per leggere i caratteri ricevuti sulla porta seriale.*

```
const
  MAXCHARS = 80;
var
  Buf: array[0..MAXCHARS] of Char;

function TApplication.IdleAction: Boolean;
var
  Result: Integer;
  NULL: TComStat absolute 0000:0000;
begin
  Result := ReadComm(ComPort, @Buf, MAXCHARS);
  if Result <= 0 then
    ReportError(GetCommError(ComPort, NULL));
  if Abs(Result) > 0 then
    (* visualizza i caratteri ricevuti *)
  IdleAction := FALSE;
end;
```

```
EV_RXCHAR  = $0001;   { carattere ricevuto e posto nella coda di input }
EV_RXFLAG  = $0002;   { ricevuto carattere di evento (EvtChar in TDCB) }
EV_TXEMPTY = $0004;   { inviato l'ultimo carattere della coda di output}
EV_CTS     = $0008;   { il segnale CTS ha cambiato stato }
EV_CTSS    = $0400;   { segnale CTS "alto" }
EV_DSR     = $0010;   { il segnale DSR ha cambiato stato }
EV_DSRS    = $0800;   { segnale DSR "alto" }
EV_RLSD    = $0020;   { il segnale RLSD ha cambiato stato }
EV_RLSDS   = $1000;   { segnale RLSD "alto" }
EV_RINGTE  = $2000;   { il segnale RING ha cambiato stato }
EV_RING    = $0100;   { riconosciuto segnale di linea }
EV_BREAK   = $0040;   { riconosciuta condizione di "break" }
EV_ERR     = $0080;   { errore di frame, di overrun o di parita' }
EV_PERR    = $0200;   { errore sulla stampante }
```

Figura 10 - Le costanti per abilitare eventi e per riconoscere se si sono verificati o meno (le costanti con valore superiore a $0200 sono state aggiunte nella API di Windows 3.1).

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