Comunicare con Windows

Gestire una porta seriale sotto DOS è impresa tutt'altro che banale. Sotto Windows non si riesce sempre a comunicare alle più alte velocità, ma, in compenso, si dispone di poche funzioni molto potenti; scrivere un programma di comunicazione risulta quindi relativamente facile; si tratta anzi di un ottimo esercizio di programmazione per eventi. Ci soffermeremo, quindi, sulla gestione sotto Windows di eventi quali la trasmissione o la ricezione di caratteri, o il verificarsi di un errore, per passare poi a vedere come riportare gli stessi principi sotto DOS, programmando per eventi con Turbo Vision

di Sergio Polini

Abbiamo praticamente terminato l'illustrazione della unit TVPRINT. Rimangono solo alcuni metodi, la cui implementazione è molto semplice. Per completezza, comunque, ve ne propongo i sorgenti nella figura 1. Il metodo *SetTab* non fa altro che assegnare un nuovo valore alla variabile d'istanza *TabWidth*; gli altri sono quasi tutti «metodi d'accesso», nel senso che consentono di prendere cognizione del valore di variabili private della classe *TPrinter*. Fa eccezione il metodo *TextLength*, che ritorna la lunghezza in pollici di una stringa e, quindi, può essere usato per operazioni di impaginazione (centratura, giustificazione a destra, ecc.).

Non chiamarmi. Ti chiamo io

Si intitola così un paragrafo del primo capitolo del libro di Petzold sulla programmazione sotto Windows. Mi piace perché sintetizza molto efficacemente l'essenza della programmazione per eventi e la sua principale differenza rispetto a stili più tradizionali.

Qualche giorno fa ho fatto visita ad un amico; stava realizzando una interessante applicazione sotto Windows, ma era giunto ad un punto morto: non riusciva a far comunicare tra loro due finestre, l'una dedicata alla visualizzazione di «cose» (perdonatemi ma, trattandosi di un'applicazione commerciale, non posso dire di più), l'altra ad un dialogo con l'utente, per determinare modalità e condizioni dell'output sulla prima.

Non so cosa sia stato, se un colpo di fortuna o la lunga abitudine a combattere con i miei bug. È successo, comunque, che ho individuato subito la radice del problema, al punto che in poco tempo, spostando qua e là alcune parti del codice, qualcosa togliendo e qualcosa aggiungendo, le due finestre comunicavano tra loro come due vecchie comari.

Sono tornato a casa incredulo: non riuscivo a credere che il mio amico, programmatore non certo alle prime armi, potesse essersi impantanato in quel modo. A ben vedere, tuttavia, la spiegazione era molto semplice: le sue finestre «chiamavano» invece di «aspettare di essere chiamate»; l'una cercava di passare il focus all'altra attraverso chiamate di procedura, invece che semplicemente smettendo di agire e aspettando che fosse Windows stesso ad attivare l'una o l'altra secondo i momenti del dialogo con l'utente.

```
procedure TPrinter.SetTab(T: Real);
begin
  TabWidth := T;
end;

function TPrinter.GetMaxX: Real;
begin
  GetMaxX := MaxX;
end;

function TPrinter.GetMaxY: Real;
begin
  GetMaxY := MaxY;
end;

function TPrinter.GetPosX: Real;
begin
  GetPosX := PosX;
end;

function TPrinter.GetPosY: Real;
begin
  GetPosY := PosY;
end;

function TPrinter.GetTab: Real;
begin
  GetTab := TabWidth;
end;

function TPrinter.GetFont(F: Word): PFont;
begin
  if F >= PD.Setup.FontList^.Count then
    GetFont := nil
  else
    GetFont := PD.Setup.FontList^.At(F);
end;

function TPrinter.GetFontCount: Word;
begin
  GetFontCount := PD.Setup.FontList^.Count;
end;

function TPrinter.TextLength(S: String): Real;
begin
  TextLength := Length(S) * CharWidth;
end;
```

Figura 1. Gli ultimi metodi della unit TVPRINT.

Ambienti come Windows o Turbo Vision *costringono* a programmare per eventi; capita, tuttavia, che talvolta ci si limiti ad assecondare un tale stile di programmazione solo finché si tratta dell'interfaccia utente: come rispondere ad un messaggio WM_PAINT, come gestire un evento *evCommand*. Si costruisce frettolosamente l'interfaccia tagliando e cucendo (ovvero con un po' di cut and paste da qualche demo), magari si prova a vedere se le finestre si aprono e il menu funziona, e poi, rassicurati, si torna ai metodi tradizionali.

Non va bene. In alcuni casi si può anche procedere così ed ottenere applicazioni perfettamente funzionanti; altre volte, però, un'accettazione solo parziale della programmazione per eventi può portare a pesanti inefficienze, o a vie senz'uscita.

Magari per telefono...

Avevo in animo di proporvi una discussione generale sulla programmazione per eventi, trattando soprattutto degli «agganci» (io li chiamo così), cioè di quei punti, presenti in ogni ambiente ad eventi che si rispetti, che consentono di inserirsi nel meccanismo generale per ottenere l'effetto giusto nel momento giusto.

C'è però MC-link, che in origine aveva un motore scritto in Turbo Pascal; ci so-

```
function BuildCommDCB(Def: PChar; var DCB: TDCB): Integer;
function ClearCommBreak(Cid: Integer): Integer;
function CloseComm(Cid: Integer): Integer;
function EnableCommNotification(Cid: Integer; Wnd: HWnd;
         cbWriteNotify, cbOutQueue: Integer): Bool;  (* Win 3.1 *)
function EscapeCommFunction(Cid, Func: Integer): Integer;
function FlushComm(Cid, Queue: Integer): Integer;
function GetCommError(Cid: Integer; var Stat: TComStat): Integer;
function GetCommEventMask(Cid, EvtMask: Integer): Word;
function GetCommState(Cid: Integer; var DCB: TDCB): Integer;
function OpenComm(ComName: PChar; InQueue, OutQueue: Word): Integer;
function ReadComm(Cid: Integer; Buf: PChar; Size: Integer): Integer;
function SetCommBreak(Cid: Integer): Integer;
function SetCommEventMask(Cid: Integer; EvtMask: Word): PWord;
function SetCommState(var DCB: TDCB): Integer;
function TransmitCommChar(Cid: Integer; Chr: Char): Integer;
function UngetCommChar(Cid: Integer; Chr: Char): Integer;
function WriteComm(Cid: Integer; Buf: PChar; Size: Integer): Integer;
```

Figura 2. Le funzioni della API di Windows per la gestione delle porte seriali e parallele.

Figura 3. Il record TDCB, *mediante il quale si possono impostare i parametri della comunicazione.*

```
TDCB = record
  Id: Byte;             { Identificativo della porta                    }
  BaudRate: Word;       { Velocita' di comunicazione                    }
  ByteSize: Byte;       { Numero di bit per byte                        }
  Parity: Byte;         { 0-4: Nessuna, pari, dispari, mark, space      }
  StopBits: Byte;       { 0,1,2 = 1, 1.5, 2                             }
  RlsTimeout: Word;     { Timeout per ricezione RLSD                    }
  CtsTimeout: Word;     { Timeout per ricezione CTS                     }
  DsrTimeout: Word;     { Timeout per ricezione DSR                     }
  Flags: Word;          { V. figura 4                                   }
  XonChar: char;        { Carattere da usare come XON                   }
  XoffChar: char;       { Carattere da usare come XOFF                  }
  XonLim: Word;         { Limite per l'invio di XON                     }
  XoffLim: Word;        { Limite per l'invio di XOFF                    }
  PeChar: char;         { Sostituto di caratteri con errore di parita'  }
  EofChar: char;        { Carattere di fine input                       }
  EvtChar: char;        { Carattere usato come segnale di evento        }
  TxDelay: Word;        { Non usato                                     }
end;
```

no alcune discussioni sulle possibili evoluzioni del sistema, sia nell'ambito della redazione che tra gli abbonati. È stato così che mi è capitato di soffermarmi sulla gestione delle porte seriali, per scoprire che si tratta di un tema che ottimanente si presta ad illustrare i come e i perché della programmazione per eventi.

Anche questa volta, come già avvenuto per le operazioni di stampa, conviene prendere le mosse da Windows; la sua API, infatti, comprende un ristretto numero di funzioni molto potenti, che semplificano enormemente la scrittura di un programma di comunicazione. Vedremo tuttavia che, per poter effettivamente operare in modo semplice, occorre riuscire a gestire in modo pulito, con idee chiare, eventi come la ricezione di un carattere. In concreto, si tratta di riuscire a cogliere questo e altri eventi senza interferire con i meccanismi di tutto l'ambiente, ma, anzi, traendone vantaggio. Si tratta, in una parola, di trovare gli «agganci» giusti. Vedremo anche che, nonostante le notevoli differenze, si deve operare in modo analogo anche quando si usa Turbo Vision.

Le funzioni della API di Windows si rivelano interessanti anche per un altro motivo; nella versione 3.1, infatti, accanto a DLL come COMMDLG o TOOLHELP, al nuovo OLE e al rinnovato DDE, ci sono novità anche per la gestione delle porte seriali. Penso in particolare alla funzione *EnableCommNotification*, che sembra essere stata aggiunta proprio per semplificare l'«aggancio».

... attraverso un modem

Nella figura 2 trovate una sintetica descrizione delle funzioni mediante le quali è possibile gestire, sotto Windows, una comunicazione attraverso le porte seriali o parallele. Non ci occuperemo delle porte parallele, per le quali, tra l'altro, molte potenzialità di quelle funzioni rimangono semplicemente inutilizzate.

Si procede innanzitutto con *OpenComm*, alla quale vanno passati come parametri il nome della porta (COM1, COM2, ecc.) e le dimensioni delle code di input e di output. Non ho disassemblato Windows, ma è facile capire a cosa servono queste code.

Un modem consente di trasmettere e ricevere dati mediante una normale linea telefonica, convertendo una sequenza di bit in suoni. Si parla di comunicazione *seriale* in quanto i dati, normali byte, vengono trasformati in sequenze di bit; non vengono inviati e ricevuti «in blocco» (otto bit alla volta), come avviene quando si manda un carattere alla stampante attraverso la porta *parallela*, ma vengono trasformati in «serie» di bit opportunamente codificate. Si parla di comunicazione seriale *asincrona* quando non si prevedono meccanismi o convenzioni estranee al normale flusso di dati per segnalare l'arrivo di un byte, ma ogni sequenza di bit corrispondente ad un carattere è codificata in modo che se ne possano riconoscere l'inizio e la fine. Ciò si ottiene tenendo alternativamente positivo e negativo il voltaggio. Tra due byte il voltaggio è negativo (si parla di condizione di *mark*); la trasmissione di un byte inizia con un bit di start, rappresentato da un voltaggio positivo (condizione di space) per un tempo uguale a quello necessario a trasmettere un bit; seguono i bit del dato (da 5 a 8, secondo i casi; il voltaggio è positivo per i bit uguali a 0, negativo per i bit

uguali a 1), un bit di «parità» (settato o meno, in genere in modo da rendere pari o dispari il numero dei bit uguali a 1 nel dato), uno o due bit di stop per segnalare la fine del byte (nel caso di dati di 5 bit, si usa un bit di stop e mezzo, cioè un voltaggio negativo per una volta e mezzo il tempo necessario per trasmettere un bit).

La conversione dei dati da byte a sequenze codificate di bit è operata dal cosiddetto ACE *(Asynchronous Communications Element)*, ovvero da un chip 8250 UART *(Uneversal Asynchronous Receiver Transmitter)* nei primi PC, 16450 o 16550 in quelli più recenti. L'ACE comprende diversi registri, tra i quali THR e RBR, destinati ad ospitare, rispettivamete, un carattere da codificare e trasmettere e un carattere ricevuto e decodificato. Per avvertire il PC di eventi come l'arrivo di un carattere (appena ricostruito in RBR) o la possibilità di inviarne uno (THR vuoto), l'ACE genera un interrupt, che viene a sua volta notificato al microprocessore mediante un *interrupt processor*, l'8259A.

Questo comporta un problema: un'applicazione non può inviare caratteri quando vuole (bisogna che il registro THR sia vuoto), né può prelevare quando vuole i caratteri in arrivo (se il registro RBR contiene un carattere quando ne arriva un altro, quello che già c'era viene perso). Sotto DOS, quindi, è necessario scrivere una routine che venga eseguita ogni volta che scatta un interrupt *(Interrupt Service Routine)*, predisponendo due code, una per i caratteri in arrivo, l'altra per i caratteri in partenza. La routine deve inserire nella coda di input i caratteri appena arrivano e prelevare un carattere dalla coda di output ogni volta che THR risulta vuoto; l'applicazione vera e propria si limiterà, quindi, a leggere e scrivere nelle due code, senza intervenire direttamente nei tempi e nei meccanismi della comunicazione. Windows ci libera dal problema, provvedendo ad allocare e gestire in modo del tutto trasparente quelle code di cui ci chiede solo la dimensione.

```
fBinary         = $0001;  { modo binario: ignora EofChar              }
fRtsDisable     = $0002;  { non usa RTS a inizio e fine collegamento  }
fParity         = $0004;  { abilitazione controllo di parita'         }
fOutxCtsFlow    = $0008;  { usa CTS per il controllo dell'output       }
fOutxDsrFlow    = $0010;  { usa DSR per il controllo dell'output       }
fDtrDisable     = $0080;  { non usa DTR a inizio e fine collegamento  }
fOutX           = $0100;  { usa XON/XOFF per il controllo dell'output }
fInX            = $0200;  { usa XON/XOFF per il controllo dell'input  }
fPeChar         = $0400;  { in caso di errore di parita', usa PeChar  }
fNull           = $0800;  { scarta i caratteri nulli                  }
fChEvt          = $1000;  { considera evento la ricezione di EvtChar  }
fDtrFlow        = $2000;  { usa DTR per il controllo dell'input       }
fRtsFlow        = $4000;  { usa RTS per il controllo dell'input       }
```

Figura 4. Il file DCBFLAGS.INC, che può essere incluso per semplificare l'avvaloramento del campo Flags *del record* TDCB.

Lo stato della porta

Se tutto va bene, *OpenComm* ritorna un intero che va poi usato per tutte le altre funzioni come identificativo della porta (ad esempio, 0 per COM1, 1 per COM2, ecc.; i codici di errore sono negativi).

Una volta aperta la porta, occorre configurare l'ACE in modo che riesca a capirsi con il modem all'altro capo della linea telefonica; dobbiamo dirgli, ad esempio, quale è la velocità di comunicazione (e quindi quanto tempo occorre per trasmettere un bit), quanti bit compongono un dato elementare, se usiamo una parità pari o dispari o nessuna parità, se si useranno uno o due bit di stop. Questo si fa scrivendo nella coppia di registri DLL e DLM *(Divisor Latch Less significant byte* e *Divisor Latch Most significant byte*, per la velocità di comunicazione) e nel registro LCR *(Line Control Register*, per bit di dati, bit di stop e parità), ma, sotto Windows, provvede a tutto la funzione *SetCommState*.

Questa vuole un parametro di tipo *TDCB* (figura 3), mediante il quale si possono impostare non solo i criteri per la codifica dei dati in sequenze di bit, ma anche altri aspetti della comunicazione.

Debbo rilevare, a questo spunto, che *TDCB* è la versione in Pascal di una struttura in C, nella quale si fa uso di quindici campi di tipo *bitfield*; non esistendo in Pascal campi di questo tipo (mediante i quali si può agire direttamente su singoli bit), si ricorre ad un unico campo *Flags* di tipo *Word*, per l'avvaloramento del quale, tuttavia, non si offre alcun aiuto. Ho dovuto verificare (a seguito di un bug che mi ha dato anche troppo da penare...) che conviene munirsi di apposite costanti, magari includendo un file come DCBFLAGS.INC, che vi propongo nella figura 4.

I primi campi di *TDCB* vanno utilizzati per precisare l'identificativo della porta, la velocità di comunicazione (espressa in *baud*, ovvero in unità di modulazione del segnale che, nei casi più comuni, e con una certa approssimazione, possono essere considerate più o meno equivalenti al *bps*, cioè al bit per secondo), il numero di bit per byte, il tipo di parità, il numero di bit di stop. Per la parità e i bit di stop si dispone delle costanti illustrate nella figura 5.

Altri campi, e alcuni bit del campo *Flags*, consentono di attribuire significato particolare ad alcuni caratteri. Settando il flag *fBinary*, ad esempio, si attiva un modo «binario» nel quale non si usa un carattere come segnale di fine input; in caso contrario, è possibile precisare quale debba essere tale carattere, mediante il campo *EofChar*. Settando il flag *fParity*, che abilita il controllo di parità, si può sostituire con *PeChar* un carattere ricevuto con un errore di parità. Settando il flag *fChEvt*, si può usare il carattere *EvtChar* come «segnale di evento»; la sua ricezione, in altri termini, genera un evento di cui si può prendere nota mediante la funzione *GetCommEventMask* (di cui riparleremo). Settando il flag *fNull*, si ottiene che vengano ignorati i caratteri nulli (ASCII 0).

Anche i campi *XonChar* e *XoffChar* possono essere utilizzati per precisare quali caratteri debbano essere usati in alcune circostanze, ma, per essere si-

```
NOPARITY        : nessun controllo di parita'
ODDPARITY       : il bit di parita' vale 1 se il numero di bit uguali a 1
                  nel dato e' pari, in modo da rendere dispari (odd) il
                  numero totale di bit uguali a 1
EVENPARITY      : il bit di parita' vale 1 se il numero di bit uguali a 1
                  nel dato e' dispari, in modo da rendere pari (even) il
                  numero totale di bit uguali a 1
MARKPARITY      : il bit di parita' vale sempre 1 (condizione di mark)
SPACEPARITY     : il bit di parita' vale sempre 0 (condizione di space)



ONESTOPBIT      : un bit di stop
ONE5STOPBITS    : un bit di stop e mezzo (per un byte di 5 bit)
TWOSTOPBITS     : due bit di stop
```

Figura 5. Le costanti simboliche per l'avvolgimento dei campi Parity *e* StopBits *del record* TDCB.

curi che siano chiare tali circostanze, dobbiamo intrattenerci un po' sull'interfaccia tra l'ACE e il modem.

Segnali e strette di mano

L'interfaccia tra ACE e modem, in un PC, è conforme allo standard RS-232-C, che definisce i termini della connessione tra un DTE *(Data Terminal Equipment,* nel nostro caso il PC) e un DCE *(Data Communication Equipment,* il modem). Fisicamente, si usano connettori a 9 o 25 pin per la trasmissione di dati e segnali. Sono sufficienti 9 pin e, quindi, cavi a 9 fili: su due di tali fili transitano i dati ricevuti e trasmessi, su altri sei i segnali illustrati nella figura 6, il nono viene usato come segnale di riferimento per il voltaggio zero.

Quei sei segnali consentono di controllare e gestire le diverse fasi della trasmissione; per usare un paragone con la comunicazione verbale, potremmo dire che servono ad assicurare che non si parli quando nessuno ascolta e che i due interlocutori non parlino contemporaneamente, ma sappiano riconoscere il momento di parlare e quello di ascoltare.

I segnali vengono implementati portando ad un voltaggio positivo o negativo i fili corrispondenti. Ad esempio, per segnalare al modem che la comunicazione può iniziare, o che sta per trasmettere qualcosa, il PC porta ad un voltaggio positivo (considerato equivalente a 0, condizione di *space)* il filo connesso ai pin DTR o RTS; un'inversione di voltaggio indica un passaggio di stato: ad esempio, da RTS a non-RTS. Per semplificare l'esposizione, diremo che si invia un segnale quando il filo corrispondente assume il voltaggio che ne denota la presenza, si abbatte un segnale quando il filo corrispondente assume un voltaggio di segno opposto; diremo che un segnale è «alto» quanto il suo filo ha il voltaggio che ne indica la presenza, è «basso» in caso contrario.

All'inizio della sessione, il PC che chiama deve inviare un segnale DTR al modem per avvertirlo che è pronto a comunicare. Il modem risponde con un DSR quando è pronto a sua volta. Prima di trasmettere dati, il PC invia il segnale RTS, al quale il modem reagisce prima inviando la portante (un segnale costituito da una singola frequenza, che verrà poi modulata per la trasmissione dei dati) al computer remoto, in modo che questo possa predisporsi alla ricezione, poi, dopo un tempo prefissato, rimandando al PC un segnale CTS. Il segnale RSLD, infine, pur non strettamente necessario, costituisce comunque conferma della presenza della portante (il segnale RI viene usato nei modem *auto-answer* per indicare che è in arrivo una chiamata).

I segnali DTR e DSR vengono in genere usati all'inizio del collegamento, per verificare che la connessione tra i due DTE è effettivamente stabilita e che, quindi, non si rischia di parlare quando nessuno ascolta.

I segnali RTS e CTS, d'altro canto, vengono in genere usati per governare singole fasi della sessione, per fare in modo che i due interlocutori parlino e ascoltino a turno, senza sovrapporre eloqui e silenzi. Si tratta, in altri termini, di rispettare le regole di qualsiasi conversazione civile, al punto che, per descrivere il coordinamento tra trasmittente e ricevente, si usa il termine *handshake* (stretta di mano).

Le regole per l'*handshake* sono le seguenti: chi vuole trasmettere, invia un RTS e aspetta un CTS prima di procedere, per smettere non appena il CTS si abbatte; chi sta ricevendo, potrebbe non riuscire a smaltire tutte le informazioni in arrivo e, quindi, abbatte il segnale RTS per chiedere una pausa, inviandolo di nuovo quando è pronto a ricevere ulteriori dati.

Queste sono le regole dell'*handshake* hardware; si può utilizzare anche un *handshake* software, realizzato mediante caratteri invece che con segnali. Si usano allo scopo i caratteri XON e XOFF (normalmente Ctrl-Q, ASCII 17, e Ctrl-S, ASCII 19): chi sta trasmettendo, smette quando riceve XOFF e riprende quando riceve XON; chi riceve, chiede una pausa con XOFF e la continuazione con XON.

I vari bit della word *Flags* consentono di scegliere tra diversi possibili usi dei segnali.

I flag *fRtsDisable* e *fDtrDisable,* se settati, provocano l'invio dei segnali RTS e DTR all'inizio del collegamento (in caso contrario, i segnali rimangono bassi); con i campi *CtsTimeout* e *DsrTimeout* si può precisare il tempo massimo di attesa, in millisecondi, dei corrispondenti segnali CTS e DSR (analoga la funzione del campo *RlsTimeout,* in relazione al segnale RLSD).

Settando i flag *fRtsFlow* e *fOutxCtsFlow* si attiva l'*handshake* hardware, con i flag *fOutX* e *fInX* si attiva l'*handshake* software; in entrambi i casi, i campi *XonLim* e *XoffLim* servono a fissare le condizioni per l'invio di segnali di pausa. Quando mancano *XoffLim* caratteri perché la coda di input sia piena, viene abbattuto RTS o inviato XOFF; si invia di nuovo RTS o XON quando il numero dei caratteri nella coda di input scende a *XonLim.*

Rimangono i flag *fDtrFlow* e *fOutxDsrFlow,* mediante i quali si può instaurare un *handshake* hardware basato su DTR/DSR invece che su RTS/CTS. Si tratta però di un metodo poco usato, anche perché alcuni modem interrompono il collegamento (riattaccano il telefono) quando DTR viene abbattuto.

In pratica, ci si può limitare a due soli valori della word *Flags*: $0301 (combinazione di *fBinary, fOutX* e *fInX*) per un *handshake* software e $4009 (combinazione di *fBinary, fOutxCtsFlow* e *fRtsFlow*) per un *handshake* hardware, assegnando, in quest'ultimo caso, 30 millisecondi a *CtsTimeout. XonLim* e *XoffLim* potranno avere un valore pari a un quarto della dimensione della rispettiva coda.

Con ciò abbiamo terminato l'esposizione dei preliminari della comunicazione. Il mese prossimo vedremo come inviare e ricevere caratteri senza interferire con i meccanismi della programmazione per eventi, ma, anzi, volgendoli a nostro favore. MS

```
DTR   (Data Terminal Ready)              : il DTE (PC) segnala al DCE (modem)
                                           che e' pronto per le comunicazioni
DSR   (Data Set Ready)                   : il DCE (modem) e' pronto per le
                                           comunicazioni
RTS   (Request To Send)                  : il DTE (PC) segnala al DCE (modem)
                                           che e' pronto a trasmettere
CTS   (Clear To Send)                    : il DCE (modem) risponde ad un RTS
                                           segnalando che e' pronto a
                                           ricevere
RLSD (Received Line Signal Detector): il DCE (modem) segnala al DTE (PC)
                                           che sta ricevendo la portante dal
                                           modem remoto (il segnale e' anche
                                           chiamato "carrier detect")
RI    (Ring Indicator)                   : usato nei modem auto-answer,
                                           indica che e' in arrivo una
                                           chiamata
```

Figura 6. I sei segnali usati per controllare e gestire una sessione di trasmissione tramite l'interfaccia RS-232-C.

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