Righe di stato

Si parte. Dopo aver ritoccato la unit TOOLBAR, vedremo ora come realizzare righe di stato capaci di coesistere con una o più barre strumenti, come inserirle in applicazioni MDI, come visualizzare messaggi esplicativi delle opzioni dei menu e dei pulsanti. Nel fare questo, modificheremo pesantemente le unit illustrate un paio d'anni fa. L'obiettivo sarà quello di definire una sorta di standard per le applicazioni MDI, estendendo ed aggiornando le funzionalità offerte dalle classi di ObjectWindows, in modo da renderle più adatte alla realizzazione di applicazioni «vere» piuttosto che di semplici demo

di Sergio Polini

Facciamo un po' di conti. Il file MDIAPP.PAS, proposto come demo di applicazioni MDI, sembrerebbe quasi una dimostrazione dei prodigi della OOP: appena 37 righe per un programma che, scritto in modo tradizionale, sarebbe circa otto volte più lungo. L'unico problema è che MDIAPP non serve assolutamente a nulla: apre finestre, le affianca, le sovrappone, le riduce a icona, le chiude. Ma queste finestre rimangono inesorabilmente vuote.

Va un po' meglio con il demo MFILEAPP.PAS (quello che trovate nella directory \BP\EXAMPLES\WIN\OWL). Grazie all'uso di unit come OSTDDLGS e OSTDWNDS, si possono creare, aprire e salvare file di testo, cercare e sostituire stringhe. Si passa però a 255 righe (+590%), senza per questo poter disporre di informazioni quali la posizione corrente del cursore o il numero dei caratteri del testo. Tali carenze, inoltre, non dipendono solo dalla mancanza di una riga di stato: semplicemente non sono disponibili. Come se non bastasse, le diverse finestre condividono tutte lo stesso segmento dati dell'applicazione, per un totale di 64K: un pesante limite ad un qualsiasi uso serio. Se poi aggiungessimo che, una volta raggiunto quel limite, nessun messaggio avverte l'utente, lasciato in muta contemplazione di file troncati o finestre vuote, potremmo solo consolarci considerando che non si tratta altro che di un demo. Ma se volessimo fare sul serio? C'è un'altra versione di MFILEAPP.PAS, quella che mostra l'uso della unit TOOLBAR (nella directory \BP\EXAMPLES\WIN\TOOLBAR); nonostante la barra strumenti ed i suoi pulsanti, però, quei limiti rimangono tutti, benché si passi a 456 righe di sorgente (un ulteriore +78%).

Si tratta di conti forse un po' noiosi; ve li propongo perché ci dicono che serve a poco un insieme di classi che costringe, per realizzare applicazioni concretamente utilizzabili, ad appesantire sempre più il file **program**, quello che dovrebbe contenere solo quanto differenzia un'applicazione da un'altra, non anche funzionalità che l'utente ha diritto di aspettarsi da qualsiasi applicazione (in unit standard).

```
unit MDIConst;

interface

const

  um_MDI                = 28672; (* base per i messaggi nelle unit *)
  um_MDIChildDestroy    = um_MDI;
  um_SetActiveChild     = um_MDI + 1;
  um_SetHelpCode        = um_MDI + 2;
  um_PaintStatusLine    = um_MDI + 3;

  cm_Print              = 24301; (* comandi "standard" *)
  cm_PrintSetup         = 24302;
  cm_Toolbar            = 24303;
  cm_StatusLine         = 24304;
  cm_Font               = 24305;
  cm_Color              = 24306;
  cm_HorizontalToolbar  = 24307;
  cm_DownHorizontalToolbar = 24308;
  cm_LeftVerticalToolbar   = 24309;
  cm_RightVerticalToolbar  = 24310;
  cm_TileVertically     = 24311;
  cm_ActivateChild      = 24312;
  cm_ChildList          = 24313;
  cm_HelpIndex          = 24314;
  cm_HelpKeyboard       = 24315;
  cm_HelpCommands       = 24316;
  cm_HelpProcedures     = 24317;
  cm_HelpHelp           = 24318;
  cm_About              = 24319;

  ids_PopupMenu         = 1000;  (* stringhe per riga di stato *)
  ids_SystemMenu        = ids_PopupMenu;
  ids_FileMenu          = ids_PopupMenu + 1;
  ids_HelpMenu          = ids_PopupMenu + 2;
  ids_MenuItem          = 2000;
  ids_ChildPopupMenu    = 3000;
  ids_ChildSysMenu      = ids_ChildPopupMenu;
  ids_ChildFileMenu     = ids_ChildPopupMenu + 1;
  ids_ChildEditMenu     = ids_ChildPopupMenu + 2;
  ids_ChildViewMenu     = ids_ChildPopupMenu + 3;
  ids_ChildOptMenu      = ids_ChildPopupMenu + 4;
  ids_ChildWinMenu      = ids_ChildPopupMenu + 5;
  ids_ChildHelpMenu     = ids_ChildPopupMenu + 6;
  ids_ChildMenuItem     = 4000;

  sl_Frame              = 0;     (* tipi di riga di stato *)
  sl_Child              = 1;
  sl_Plain              = 2;

implementation

end.
```

Figura 1 - Una unit con le costanti per «qualsiasi» applicazione MDI.

Per qualche costante in più

Non parleremo, quindi, solo di righe di stato, ma di tutto un nuovo impianto per le applicazioni MDI.

La figura 1 contiene il listato di una unit MDICONST che, pur se non utilizzata direttamente dalla riga di stato, verrà usata dalle altre unit che vi proporrò.

Si inizia con quattro messaggi che ho ripreso da quella unit STRIPES che vi avevo proposto nel numero di gennaio 1992 e su cui torneremo quando esamineremo le nuove unit MDI.

Segue la dichiarazione di una ventina di costanti per comandi che mi sembra ragionevole considerare presenti in qualsiasi applicazione: la stampa e l'impostazione della stampante, la scelta del carattere e del colore, la visualizzazione o meno della barra strumenti e della riga di stato, l'orientamento della riga di stato, l'affiancamento in verticale – oltre che in orizzontale – delle finestre aperte, la scelta di una finestra dall'omonimo menu, le opzioni standard del menu di help.

Ne vedremo l'uso in seguito, nel demo di un'applicazione completa e concretamente utilizzabile.

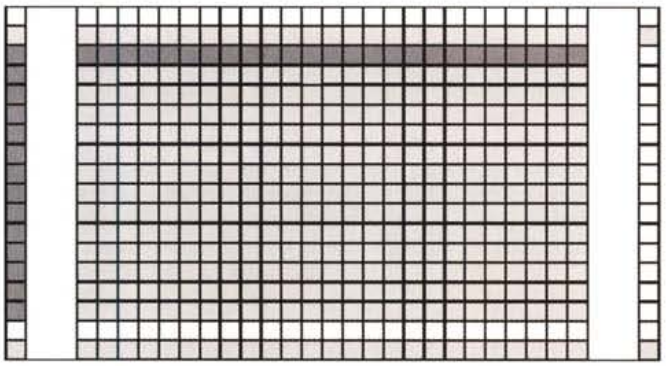
Si prosegue con costanti dedicate ad identificare le stringhe che dovranno essere visualizzate sulla riga di stato quando l'utente si posizionerà su un menu. Non si tratta, in realtà, di costanti strettamente necessarie; tra l'altro, presuppongono una struttura del menu rigorosamente aderente allo standard (un menu principale con opzioni *File, Modifica, Visualizza, Opzioni, Finestre, Help*). Sono lì in quanto le ho trovate molto utili nella preparazione della tabella di stringhe da inserire nel file di risorse: meglio associare una stringa ad una costante che identifica il menu *File* piuttosto che al numero 1001 o 3001. Nulla vieta, peraltro, di dichiarare costanti diverse in un'applicazione. Si tratta solo di tenere presente, come vedremo, che i singoli menu vengono identificati secondo il loro numero ordinale (primo, secondo, terzo, e così via).

La unit termina con la dichiarazione di tre costanti per tre diversi «tipi» di riga di stato. Nell'intento di dare la maggiore flessibilità possibile all'impianto, infatti, ho previsto che la riga di stato possa avere un aspetto ed un contenuto diversi secondo che sia aperta o meno almeno una finestra; inoltre, analogamente a quanto accade in applicazioni quali Word per Windows o lo stesso Borland Pascal, la riga di stato potrà assumere un aspetto ancora diverso quando su di essa verrano mostrati messaggi esplicativi delle opzioni dei menu, o comunque massaggi non condizionati da una sua suddivisione in campi (ad esempio, «Salvataggio in corso» durante la scrittura su disco del contenuto della finestra attiva).

Figura 2 - L'interfaccia della unit STATLINE.

```
unit StatLine;
{$R STATLINE.RES}
interface

uses
  Wintypes, Winprocs, Objects, OWindows;

const { riprese dalla unit Toolbar }
  am_CalcParentClientRect = wm_User + 120;
  tbHidden                = $00;
  AllowRepaint            = 1;

type
  PStatusField = ^TStatusField;
  TStatusField = object(TObject)
    constructor Init(XFrom, TxtLen, XTo: Integer);
    destructor Done; virtual;
    procedure SetText(T: PChar);
  private
    X1: Integer;
    X2: Integer;
    Len: Integer;
    Txt: PChar;
    procedure Paint(PaintDC, MemDC: HDC; Left, Middle, Right: HBitmap);
  end;

  PStatusLine = ^TStatusLine;
  TStatusLine = object(TWindow)
    constructor Init(AParent: PWindowsObject; FrameFields: PCollection;
                     Orient: Word);
    destructor Done; virtual;
    procedure SetupWindow; virtual;
    function  GetOrientation: Word;
    procedure SetOrientation(NewOrient: Word);  virtual;
    procedure SwitchTo(NewFields: PCollection);
    procedure SetText(T: PChar);
    procedure WMEraseBkgnd(var Msg: TMessage);
      virtual wm_First + wm_EraseBkgnd;
    procedure Paint(PaintDC: HDC; var PaintInfo: TPaintStruct); virtual;
    procedure AMCalcParentClientRect(var Msg: TMessage);
      virtual wm_First + am_CalcParentClientRect;
  private
    Orientation: Word;
    Txt: PChar;
    Fields: PCollection;
    Rect: TRect;
    LF: TLogFont;
    WhitePen: HPen;
    Left, Middle, Right, MemBM: HBitmap;
  end;
```

Figura 3 - Una rappresentazione delle tre bitmap contenute nel file STATLINE.RES, nell'ordine: Left *(1x18),* Middle *(qui 25x18, ma 64x18 nel file) e* Right *(1x18).*



I campi

Trovate nella figura 2 l'interfaccia della unit STATLINE. Viene usato un file di risorse STATLINE.RES, comprendente tre bitmap *Left, Middle* e *Right*, rappresentate nella figura 3. *Left* e *Right* servono per i lati sinistro e destro di un «campo» della riga di stato ed hanno una dimensione di 1x18 (larghezza x altezza); *Middle* ha una dimensione 64x18, ma viene rappresentata nella figura come un rettangolo 25x18. La unit riprende alcune costanti della unit TOOLBAR. Un approccio più rigoroso richiederebbe la preparazione di una unit «base», contenente sia le costanti comuni, sia la dichiarazione di una classe astratta da cui derivare tutte le classi destinate ad essere istanziate in oggetti che, inseriti nella *client window* di un'applicazione MDI o nella *client area* di una finestra, ne occupino una parte in modo da non pregiudicare il comportamento di altre *child window* (ad esempio, un *ribbon*). Ho preferito, tuttavia, evitare di intervenire troppo pesantemente sul codice di una unit già disponibile.

Vengono dichiarate due classi, *TStatusField* e *TStatusLine*.

```
implementation

uses Strings;

var
  SLWindow: HWnd;

constructor TStatusField.Init(XFrom, TxtLen, XTo: Integer);
begin
  inherited Init;
  X1 := XFrom;
  X2 := Xto;
  Len := TxtLen;
  GetMem(Txt, Len+1);
  FillChar(Txt^, Len+1, 0);
end;

destructor TStatusField.Done;
begin
  FreeMem(Txt, Len+1);
  inherited Done;
end;

procedure TStatusField.SetText(T: PChar);
var
  R: TRect;
  XL, XR: Integer;
begin
  StrLCopy(Txt, T, Len);
  XL := Abs(X1);
  XR := Abs(X2);
  SetRect(R, XL+5, 3, XR-1, 16);
  InvalidateRect(SLWindow, @R, True);
end;

procedure TStatusField.Paint(PaintDC, MemDC: HDC;
                             Left, Middle, Right: HBitmap);
var
  XL, XR: Integer;
begin
  if X1 >= 0 then begin
    XL := X1;
    SelectObject(MemDC, Left);
    BitBlt(PaintDC, XL, 0, 1, 18, MemDC, 0, 0, SrcCopy);
  end
  else XL := -X1-1;
  if X2 >= 0 then begin
    XR := X2;
    SelectObject(MemDC, Right);
    BitBlt(PaintDC, XR, 0, 1, 18, MemDC, 0, 0, SrcCopy);
  end
  else XR := - X2;
  SelectObject(MemDC, Middle);
  StretchBlt(PaintDC, XL+1, 0, XR-XL-1, 18, MemDC, 0,0,64,18, SrcCopy);
  TextOut(PaintDC, XL+5, 3, Txt, StrLen(Txt));
end;
```

Figura 4 - L'implementazione della classe TStatusField.

La classe *TStatusField* (figura 4), derivata da *TObject*, è destinata ad avere come istanze i «campi» di una riga di stato. Mediante il constructor si stabiliscono l'ascissa del bordo sinistro del campo, il numero massimo dei caratteri della stringa che in esso dovrà essere visualizzata, l'ascissa del bordo destro. Al momento dell'inizializzazione, si alloca memoria per una stringa vuota, che verrà poi rilasciata dal destructor.

Il metodo *SetText* permette di sostituire in un qualsiasi momento una data stringa a quella, inizialmente vuota, assegnata alla variabile d'istanza privata *Txt*, provocando contestualmente un *repaint* della zona del campo occupata dal testo. Per evitare inutili complicazioni, i campi della riga di stato non sono finestre, ma zone dell'unica finestra associata alla riga di stato; la funzione *InvalidateRect*, quindi, non ha come primo parametro l'*handle* di una finestra associata al campo, ma quello della finestra della riga di stato. Per evitare, inoltre, di appesantire la classe *TStatusField* con un'ulteriore variabile d'istanza, all'inizio della sezione di implementazione della unit si dichiara una variabile *SLWindow* che, come vedremo, viene avvalorata dal metodo *TStatusLine.SetupWindow*. La soluzione presuppone, ovviamente, che in un'applicazione vi sia una sola riga di stato (più esattamente: che vi sia una sola zona – e sempre quella – della *client window* destinata ad ospitare una riga di stato, anche se formato e contenuto di questa possono variare).

Il metodo *Paint* vuole come argomenti due *device context* e tre bitmap. Quanto a queste ultime, si tratta di quelle illustrate nella figura 3, caricate da *TStatusLine* e da questa passate ai suoi campi. Ogni campo conosce le ascisse del suo bordo sinistro e del suo bordo destro, rappresentate, di norma, mediante le bitmap *Left* e *Right* (la bitmap *Middle* viene usata per disegnare lo parte di un campo compresa tra i due bordi sinistro e destro). Dico «di norma» perché è possibile definire campi contigui, non separati dai bordi destro e sinistro. Ad esempio, è possibile sia definire un unico campo destinato a visualizzare la data e l'ora corrente, sia prevedere due campi, uno per la data e l'altro per l'ora, ma farli apparire in un unico rettangolo della riga di stato: un bordo sinistro a sinistra del campo «data», il valore corrente della data, seguito immediatamente dal valore dell'ora, a sua volta seguito dal bordo destro; è possibile, quindi, omettere la rappresentazione del bordo a destra della data e del bordo a sinistra dell'ora, evitando, in questo modo, di suddividere la riga di stato in un numero eccessivo di campi e mantenendo una contiguità visiva tra campi logicamente correlati. Ciò si può ottenere indicando valori negativi per l'ascissa destra o sinistra di un campo. In pratica: supponendo di volere data o ora in campi distinti, questi andranno definiti inizializzando, ad esempio, il campo «data» con 25, 8, 80 (ascissa del bordo sinistro, lunghezza del testo, ascissa del bordo destro), il campo «ora» con 85, 5, 140; volendo mostrare invece data e ora in un unico rettangolo della riga di stato, mantenendo comunque distinti i due campi (in quanto aggiornati con ritmi evidentemente molto diversi), questi andranno definiti indicando 25, 8, –80 per la data e –80, 5, 135 per l'ora.

Ecco il motivo per cui il metodo Paint controlla il segno delle variabili d'istanza *X1* e *X2* (le ascisse dei bordi sinistro e destro) e traccia le bitmap *Left* e *Right* solo se il valore corrispondente è positivo.

L'una o l'altra delle due bitmap, se va visualizzata, viene prima selezionata in *MemDC*, un *memory device context* creato dal metodo *TStatusLine.Paint*, quindi direttamente copiata sul *device context* rappresentato da *PaintDC* mediante la funzione *BitBlt*. La bitmap *Middle*, invece, viene adattata alle dimensioni del campo mediante la funzione *StretchBlt*. Una volta disegnati, mediante le tre bitmap, i contorni e lo sfondo del campo, viene scritto anche il testo.

Quanto a *PaintDC*, il nome potrebbe far credere che si tratta del *device context* associato alla zona dello schermo occupata dal campo. In realtà, sebbene potrebbe essere così, così non è: anche *PaintDC* è un *memory device context*, cioè una zona di memoria che, pur avendo gli stessi attributi di un *device context* connesso allo schermo, non produce alcun output su video.

Output senza tremolio

Per quelli (come me) che preferiscono l'inglese ad opinabili traduzioni italiane, chiarisco subito che «tremolio» è la migliore traduzione che sono riuscito a trovare per *flickering*. Cosa voglia dire *flickering* è presto detto.

Se il campo di una riga di stato venisse visualizzato in modo «normale», se cioè nel parametro *PaintDC* del me-

todo *TStatusField.Paint* si passasse un normale *device context*, accadrebbe questo: ogni richiesta di ridisegnare la zona della riga di stato occupata da un campo darebbe luogo all'invio, da parte di Windows, dei messaggi *wm_EraseBkgnd* e *wm_Paint*; il primo provocherebbe la cancellazione di quella zona mediante il suo riempimento con il colore di sfondo, il secondo la riscrittura del suo contenuto. Risultato: l'alternanza tra cancellazione e riscrittura sarebbe percettibile, ogni cosa scritta apparirebbe continuamente assente e presente, e, quindi, presente ma «tremolante».

Per evitare tutto ciò, si deve intercettare il messaggio *wm_EraseBkgnd* in modo che non produca effetto alcuno e, al tempo stesso, provvedere a ridisegnare in memoria (in un *memory device context*) la zona che interessa, sfondo compreso, per sovrapporla in un colpo solo a quanto appariva sullo schermo.

La figura 5 mostra i metodi di *TStatusLine* che provvedono al tutto.

Il metodo *SetupWindow*, oltre ad avvalorare la variabile *SLWindow*, inizializza la variabile d'istanza *MemBM* creando una bitmap compatibile con il *device context* della finestra destinata ad ospitare la riga di stato. Si assumono come dimensioni iniziali valori sostanzialmente fittizi; la bitmap verrà infatti adattata alle dimensioni effettive della riga di stato dal metodo *AMCalcParentClientRect*, che vedremo nel prossimo numero.

Il metodo *WMEraseBkgnd* non fa nulla, in quanto la riga di stato non deve rispondere al corrispondente messaggio.

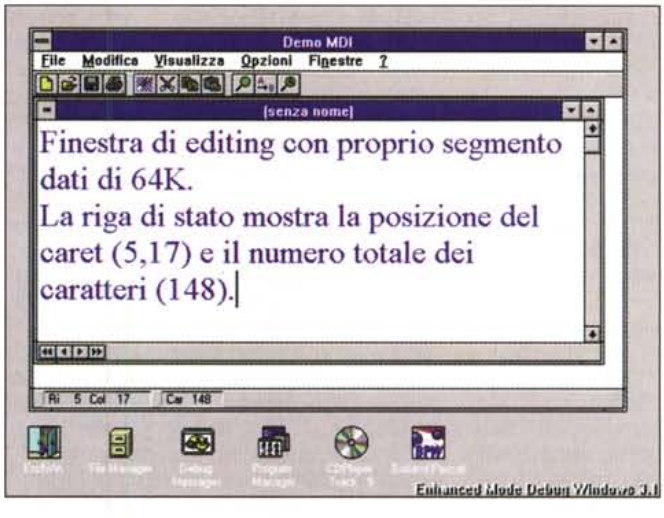
Il metodo *Paint* seleziona la bitmap *MemBM* in un *memory device context* compatibile con quello «fisico», *MemDC*, che viene subito riempito usando un *brush* di colore grigio, coerente con le scale di grigio usate per le bitmap *Left*, *Middle* e *Right*; si provvede così a disegnare lo sfondo della riga di stato.

Si seleziona quindi in *MemDC* una *pen* di colore bianco (la variabile *WhitePen* è inizializzata dal constructor mediante la funzione *GetStockObject*) per tracciare una riga bianca immediatamente sotto il bordo superiore della finestra, come già abbiamo visto nella unit TOOLBAR. Si seleziona poi un font corrispondente a quello definito (dal constructor) mediante la variabile d'istanza *LF* e si prepara l'output del testo in modo che non alteri il colore di sfondo. Completati i preliminari, si esamina la variabile d'istanza *Fields*; se questa vale **nil** (se, cioè, non vanno disegnati distinti campi), si scrive il testo puntato dalla variabile *Txt*; altrimenti si crea un ulteriore *memory device context*, *FldDC*, a beneficio dei diversi campi; questi vengono disegnati mediante la chiamata, per ognuno, del suo metodo *Paint*, passando come parametri *FldDC* come *device context* «di appoggio» e *MemDC* come *device context* di output. Fino a questo punto, ogni operazione di output è avvenuta su un *device context* virtuale, esistente solo in memoria; l'output su schermo viene realizzato copiando *MemDC* su *PaintDC* tutto in una volta, sfondo, bordi dei campi e testo compresi. Risultato: una riga di stato *flicker-free*. Nel prossimo appuntamento completeremo l'illustrazione della riga di stato, vedendo anche come ottenere le informazioni che è lecito aspettarsi ci vengano da essa comunicate, quali una breve spiegazione dell'opzione di un menu o la posizione del *caret* in una finestra di editing. E altre cose ancora. MC

Figura 5 - I metodi di TStatusLine *che si occupano della visualizzazione della riga di stato.*

```
procedure TStatusLine.SetupWindow;
var
  DC: HDC;
begin
  inherited SetupWindow;
  SLWindow := HWindow;
  DC := GetDC(HWindow);
  MemBM := CreateCompatibleBitmap(DC, 20, 20);
  ReleaseDC(HWindow, DC);
end;

procedure TStatusLine.WMEraseBkgnd(var Msg: TMessage);
begin
end;

procedure TStatusLine.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);
var
  MemDC, FldDC: HDC;
  OldBM: HBitmap;
  OldPen: HPen;
  BrBkgnd: HBrush;
  Font, OldFont: HFont;
  R: TRect;
procedure PaintIt(Item: PStatusField); far;
  begin
    Item^.Paint(MemDC, FldDC, Left, Middle, Right);
  end;
begin
  GetClientRect(HWindow, R);
  MemDC := CreateCompatibleDC(PaintDC);
  OldBM := SelectObject(MemDC, MemBM);
  BrBkgnd := CreateSolidBrush(RGB(192, 192, 192));
  FillRect(MemDC, R, BrBkgnd);
  DeleteObject(BrBkgnd);
  OldPen := SelectObject(MemDC, WhitePen);
  MoveTo(MemDC, 0, 0);
  LineTo(MemDC, Attr.W + 1, 0);
  SelectObject(MemDC, OldPen);
  Font := CreateFontIndirect(LF);
  OldFont := SelectObject(MemDC, Font);
  SetBkMode(MemDC, TRANSPARENT);
  if Fields = nil then begin
    if Txt <> nil then
      TextOut(MemDC, 5, 3, Txt, StrLen(Txt));
  end
  else begin
    FldDC := CreateCompatibleDC(PaintDC);
    Fields^.ForEach(@PaintIt);
    DeleteDC(FldDC);
  end;
  BitBlt(PaintDC, 0, 0, R.Right, R.Bottom, MemDC, 0, 0, SrcCopy);
  SelectObject(MemDC, OldFont);
  SelectObject(MemDC, OldBM);
  DeleteDC(MemDC);
  DeleteObject(Font);
end;
```

Figura 6 - Un demo MDI con barra strumenti e riga di stato per la finestra principale e con finestre «figlie» dotate ognuna di propria barra strumenti.



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