Hook, subclassing e... WordPad

di Alessandro Scotti

Uno degli accessori più utili che si trovano nella versione standard di Windows 95 è sicuramente WordPad, un word processor che può leggere e scrivere file anche in formato Word per Windows 6.0. La somiglianza con Word però non va molto più in là della compatibilità tra documenti, e in WordPad si trovano solo alcune delle funzioni più elementari per il trattamento dei testi. Dal momento che uso WordPad piuttosto assiduamente, mi sono trovato di recente ad affrontare il problema di come ottenere delle informazioni sul testo di un documento, per esempio quante parole contiene, quanti paragrafi e così via. La prima idea che mi è venuta in mente è stata quella di salvare il documento in modo testo, in modo da poterlo poi elaborare con un semplice programma DOS, ma questa ed analoghe soluzioni presentano l'inconveniente di essere troppo laboriose e scomode per un utilizzatore abituale. Ovviamente la soluzione ideale sarebbe quella di avere un apposito comando all'interno di WordPad, ma sarebbe necessario disporre dei sorgenti dell'applicazione per poterla modificare... o no?

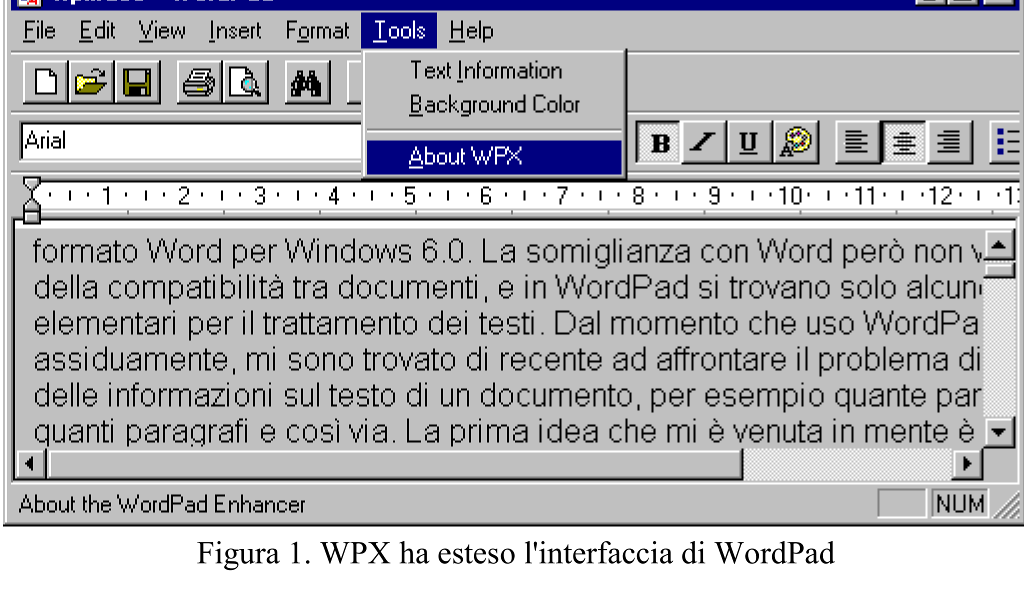

Figura 1. WPX ha esteso l'interfaccia di WordPad

Primi passi

WordPad è essenzialmente l'interfaccia verso l'utente per un controllo standard di Windows 95, chiamato RichEdit. A differenza dei normali controlli di tipo Edit, che possono gestire solo semplici stringhe di caratteri, un controllo RichEdit supporta testo "arricchito" da altre informazioni, come font e oggetti OLE. Torneremo su questo argomento in seguito, l'importante per ora è ricordare che un RichEdit fornisce il "motore" per trattare e visualizzare questo tipo di testi, ma l'interfaccia con le funzioni più avanzate deve essere fornita da una applicazione vera e propria. Il compito di WordPad consiste dunque nel permettere all'utente di interagire con il controllo RichEdit e viceversa.

Facciamo ora qualche esperimento per modificare l'interfaccia di WordPad. Una volta ottenuto l'handle della finestra con FindWindow (la classe è "WordPadClass") possiamo ricavare l'handle del menu principale con GetMenu ed inserire una nuova voce con InsertMenu. Funziona, ma non è molto utile, dal momento che non possiamo poi ricevere il corrispondente messaggio WM_COMMAND. Proviamo allora a parlare con il controllo RichEdit contenuto all'interno di WordPad, il cui handle si ottiene con EnumChildWindows cercando una finestra di classe "RICHEDIT". Qui le cose vanno un po' meglio: è possibile eseguire un certo numero di operazioni inviando messaggi al controllo, perlomeno fino al momento in cui si invia WM_GETTEXT per riceverne il contenuto. A questo punto infatti, nel migliore dei casi Windows 95 ci avviserà che WordPad ha effettuato una operazione illegale e che verrà terminato. Il problema è che il controllo RichEdit cerca di copiare il testo all'indirizzo che ha ricevuto insieme al messaggio WM_GETTEXT, ma questo indirizzo è relativo allo spazio di indirizzamento del programma che ha inviato il messaggio! La separazione degli address space nelle applicazioni a 32 bit fa sì che risulti molto difficile scambiare puntatori in modo affidabile. Il bello è che il programma che invia il messaggio WM_GETTEXT continua a funzionare tranquillamente, mentre l'applicazione che l'ha ricevuto viene terminata in seguito all'eccezione.

I due precedenti esperimenti dimostrano che per effettuare operazioni di qualche utilità è necessario trovarsi all'interno dello spazio di indirizzamento di WordPad. Naturalmente quest'ultimo non è consapevole del fatto che stiamo cercando di fargli un favore estendendone le funzionalità, perciò non dovremo aspettarci alcun tipo di aiuto!

Come accedere all'address space di WordPad

In Win32, ci sono diversi sistemi per fare in modo che una DLL venga caricata nell'address space di un particolare programma. Alcuni di essi sono disponibili solo su Windows NT, mentre altri richiedono l'uso delle API di debug o la modifica del Registry. Il metodo che ho usato si basa sugli hook (vedi "DLL, hook e subclassing in Win32" di Dino Esposito su *Computer Programming* n. 41, Novembre 1995). Brevemente, istallare un hook di sistema significa chiedere a Windows di invocare una funzione quando si verifica un particolare evento o una classe di eventi. Nel trasferire questo meccanismo da Win16 a Win32 i progettisti si sono trovati di fronte al solito problema degli spazi di indirizzamento: se la funzione hook si trova nel processo A ma l'evento viene generato dal processo B, cosa bisogna fare? Non potendo ogni volta effettuare un context switch ad A e poi ancora a B (il passaggio da un processo ad un altro è una operazione molto lenta, ed inoltre rimane il problema di eventuali indirizzi e puntatori legati all'evento che non sarebbero validi nello spazio di A), Win32 deve trasferire la funzione collegata all'evento nello spazio di B, in modo da poterla chiamare senza problemi nello stesso contesto in cui si è verificato l'evento. Per questo motivo le funzioni di hook devono risiedere in una DLL, che verrà inserita nello spazio di indirizzamento del processo che ha generato l'evento alla prima occasione utile. Una importante osservazione è che l'intera DLL viene inserita nel processo ospite, e non solo la funzione associata all'evento: fare altrimenti avrebbe reso molto difficile scrivere funzioni di hook efficienti e limitato la compatibilità con l'ambiente Win16.

Benché il progetto originale non lo richiedesse, per illustrare meglio l'uso degli hook ho posto un ulteriore obiettivo: modificare l'interfaccia utente di tutte le istanze di WordPad presenti sul sistema. Soddisfare questo requisito significa essere in grado di:
- riconoscere le istanze di WordPad presenti al momento del lancio del programma;
- intercettare le istanze di WordPad avviate mentre il programma di controllo è in esecuzione.

Il programma di controllo (WPX da ora in poi) identifica le istanze di WordPad già avviate controllando la classe delle finestre attive, che possono essere elencate con EnumWindows. Se vengono effettivamente trovate delle finestre appartenenti alla classe registrata da WordPad viene istallato un hook di tipo WH_GETMESSAGE, grazie al quale la funzione di hook viene invocata ogni volta che un programma accede alla propria coda eventi. Quando WordPad chiama una funzione come GetMessage per prelevare un messaggio, Windows si accorge che c'è un hook istallato e se non è già presente inserisce la DLL contenente la funzione di hook nello spazio di indirizzamento di WordPad. Tutte le funzioni di hook usate da WPX si trovano in una DLL chiamata WPXDLL, perciò il risultato è quello di avere questa DLL correttamente inserita nell'address space di tutte le istanze di WordPad presenti. In effetti, è abbastanza probabile che WPXDLL venga caricata nello spazio di ogni processo, ma questo non è un problema perché la funzione di hook effettua pochissime operazioni e l'overhead da essa causato risulta trascurabile.

Riconoscere l'avvio di ulteriori istanze di WordPad quando WPX è in esecuzione richiede l'utilizzo di un altro tipo di hook, WH_SHELL, che viene invocato ogni volta che una finestra "principale" viene creata o distrutta (sebbene in Windows 95 la funzione di hook venga invocata anche in seguito ad altri eventi, per esempio l'attivazione di una finestra, sfruttare questa caratteristica significa rendere il codice non portabile su Windows NT senza ricevere in cambio un vantaggio in qualche modo significativo). Anche in questo caso, non appena un programma crea la sua prima finestra Windows inserisce la DLL che contiene la funzione di hook nello spazio di indirizzamento del processo interessato all'evento: a noi non rimane altro che controllare se la finestra appena creata appartiene o no a WordPad.

Come si vede, in entrambi i casi siamo riusciti ad inserire WPXDLL nell'address space di WordPad, e questo è un significativo passo avanti. Il nostro prossimo obiettivo è ora quello di effettuare il subclassing della finestra principale di WordPad, ovvero fare in modo che tutti i messaggi inviati a WordPad vengano intercettati da una nostra funzione.

Subclassing, finalmente!

Le funzioni di hook descritte nel precedente paragrafo, non servono solo ad accedere allo spazio del processo di WordPad, ma possono essere sfruttate anche per altri compiti. In particolare l'hook di tipo WM_SHELL risulta utilissimo non solo per capire quando viene avviata una nuova istanza di WordPad, ma anche per accorgersi quando una istanza viene terminata. Il compito della funzione di hook (wpxShellHook in WPXDLL.C) è molto semplice: se Windows chiama l'hook indicando che una finestra è stata creata, la funzione controlla la classe della finestra e se questa viene riconosciuta come una finestra principale di WordPad ne effettua il subclassing ed invia un messaggio al programma WPX, in modo che il numero globale delle finestre di WordPad attive sia noto in qualsiasi momento. Se invece l'hook viene invocato perché una finestra sta per essere distrutta, la funzione controlla ancora se la finestra appartiene a WordPad ed in caso affermativo invia un apposito messaggio a WPX, che decrementa il contatore delle finestre attive. Quando il contatore scende a zero significa che non ci sono più istanze di WordPad attive ed il programma si chiude automaticamente.

Leggermente differente è il funzionamento dell'hook WH_GETMESSAGE: tutti i messaggi vengono ignorati tranne PM_SUBCLASSWINDOW, un messaggio privato e definito in modo da evitare conflitti che provoca il subclassing della finestra che lo riceve. Naturalmente WPX invia questo messaggio solo alle finestre principali di WordPad, e per la precisione a tutte le finestre che, essendo già state create al momento del lancio di WPX, non potrebbero essere riconosciute dall'hook WH_SHELL. Per questo motivo, se non ci sono finestre di WordPad già attive quando WPX effettua il controllo iniziale, l'hook WH_GETMESSAGE non è necessario e non viene istallato, in modo da minimizzare ulteriormente l'utilizzo di risorse.

Riassumendo: subito dopo la partenza WPX elenca le finestre principali presenti in Windows e se ne trova qualcuna appartenente a WordPad istalla un hook di tipo WH_GETMESSAGE ed invia un messaggio PM_SUBCLASSWINDOW alle finestre di cui vuole effettuare il subclassing. Inoltre un hook di tipo WH_SHELL permette sia di riconoscere la creazione di nuove finestre WordPad, di cui viene immediatamente effettuato il subclassing, che la distruzione di finestre già esistente. In entrambi i casi il subclassing può essere effettuato in sicurezza perché la DLL contenente le funzioni di hook viene inserita da Windows nello spazio di indirizzamento del processo che ha provocato l'invocazione di una qualsiasi funzione di hook.

Bene... finalmente siamo riusciti ad entrare nell'address space di qualsiasi istanza di WordPad e ad effettuare subclassing della finestra principale! Cerchiamo ora qualche sistema per modificare l'interfaccia utente e rendere accessibili le funzioni che vogliamo aggiungere.

Spic come noi

Dopo aver effettuato il subclassing della finestra, WPXDLL invia a WPX il messaggio PM_INSERTWINDOW, che WPX utilizza per incrementare il contatore delle finestre attive. WPX risponde con un ulteriore messaggio PM_INITGUI, che viene spedito a WordPad per indicare che l'interfaccia utente deve essere modificata. Il messaggio può essere inviato tranquillamente alla finestra di WordPad, perché dal momento che ne è stato già effettuato il subclassing esso verrà intercettato dalla nostra funzione WndProc e opportunamente gestito (WordPad rimane l'unico a non sapere cosa gli sta accadendo...).

Modificare il menu principale è per WPXDLL un gioco da ragazzi (vedi "xxx" di Dino Esposito su CP xx, xxx 1996) e le nuove funzioni vengono abilitate controllando se i messaggi WM_COMMAND inviati alla finestra si riferiscono al nuovo menu. Nonostante l'apparente semplicità bisogna però essere consapevoli di alcuni comportamenti che dipendono dal modo in cui è stato implementato WordPad. Per prima cosa, gli elementi del menu principale possono essere indirizzati in due modi: con la posizione e con l'handle del corrispondente menu popup. Se WordPad utilizzasse il primo metodo, sarebbe per noi più difficile inserire il nuovo menu in una posizione qualsiasi, ma per fortuna ciò non accade e ci siamo risparmiati un po' di lavoro. Inoltre le librerie MFC (Microsoft Foundation Classes) con cui è stato scritto WordPad disabilitano tutte le voci di menu per le quali non è stata registrata una funzione di gestione, figuriamoci quindi i menu inseriti da altri programmi a run-time! Insomma, un altro messaggio da intercettare: quando la nostra WndProc riceve il messaggio WM_INITMENUPOPUP controlla se si tratta del nuovo menu ed in caso affermativo ritorna subito senza lasciare a WordPad la possibilità di combinare guai.

Bene, dovremmo disporre ora di un menu abilitato e funzionante, no? Quasi. Scorrendo le voci dei menu "originali", si nota come la linea di stato venga aggiornata con una breve descrizione del comando correntemente selezionato, una caratteristica che andrebbe estesa anche al nostro menu se vogliamo fare in modo che l'integrazione sia senza macchia. La parte più difficile consiste nel trovare l'handle della status bar, perché la corrispondente finestra è di classe "AfxControlBar" così come molte altre finestre all'interno di WordPad (si tratta di una classe standard delle MFC). Non potendo nemmeno distinguere le varie finestre per nome o titolo, che non hanno, bisogna caratterizzare la finestra di stato in modo diverso. Così, per WPXDLL la status bar è la finestra di classe "AfxControlBar" che si trova più in basso rispetto alle altre: è una regola empirica, ma fino ad ora ha sempre funzionato! Individuata finalmente la sospirata finestra, non resta che intercettare i messaggi WM_MENUSELECT che Windows invia quando viene selezionata una nuova voce di menu: se il corrispondente menu appartiene a WordPad il messaggio viene lasciato passare, altrimenti si sceglie il messaggio appropriato in base alla voce di menu selezionata e lo si invia alla status bar con il messaggio WM_SETTEXT.

Nel caso che qualcuno si chieda (a ragione) come sono stati ottenute le precedenti informazioni su finestre e messaggi, la risposta è molto semplice: con una piccola utility chiamata Spy e fornita con il Win32 SDK. Questo programma permette di selezionare una finestra qualsiasi tra quelle attive sul desktop (utilissimo anche per trovare i nomi delle corrispondenti classi) e di visualizzare in tempo reale i messaggi da essa ricevuti. E' davvero sorprendente quante cose si possono apprendere su un'applicazione osservandone i messaggi! Per di più, Spy usa gli hook e viene rilasciata completa di sorgenti: sembrerebbe quindi un ottimo punto di partenza per un programma come WPX, ma come al solito è meglio procedere con cautela e verificare anche codice che data la provenienza si dovrebbe dare per garantito. In particolare all'interno della funzione di gestione dell'hook WH_GETMESSAGE, troviamo che CallNextHookEx viene chiamata passando NULL al posto dell'handle HHOOK restituito da SetWindowsHookEx, dal momento che "il primo parametro viene ignorato". Questa affermazione non si trova nella documentazione ufficiale, ed inoltre risulta falsa per altri tipi di hook: se non si utilizza l'handle corretto in un hook di tipo WH_SHELL, ad esempio, la taskbar di Windows 95 non riesce più a tenere traccia delle finestre attive.

Risorse condivise

Ovviamente Spy funziona benissimo, ma in WPXDLL ho deciso di attenermi alla documentazione ufficiale. Questa scelta comporta la necessità di trovare un meccanismo per rendere accessibili a tutte le istanze della DLL gli handle HHOOK da utilizzare con CallNextHookEx, altrimenti disponibili al solo processo che ha istallato gli hook con SetWindowsHookEx. La soluzione più immediata è quella di rendere globali le corrispondenti variabili, ma in Win32 ogni istanza di una DLL ha per default un proprio segmento dati, piuttosto che condividere un unico segmento dati globale come avviene in ambiente Win16. Sebbene sia possibile forzare la creazione di una zona di dati condivisa dalle varie istanze di una DLL, il mio parere è quello di evitare di ricorrere a questo espediente se non ce n'è un'autentica necessità, perché in generale la gestione della shared memory in ambiente multitasking è potenzialmente fonte di problemi se non effettuata con appropriati meccanismi di sincronizzazione. Per questo motivo le due variabili che devono essere condivise da tutte le istanze di WPXDLL, ovvero gli handle delle funzioni di hook, vengono memorizzate dal programma WPX, che le rende accessibili mediante particolari messaggi. Quando WPXDLL viene inizializzata con il codice DLL_PROCESS_ATTACH, ad indicare che la DLL sta per essere inserita in un nuovo processo, ottiene l'handle della finestra di WPX chiamando FindWindow: se l'handle è diverso da NULL, gli handle delle funzioni di hook vengono allora ricavati inviando con SendMessage i messaggi PM_GETMSGHOOK e PM_GETSHELLHOOK a WPX, e memorizzati in variabili globali per la specifica DLL.

DLL ed entry point

Una volta stabiliti gli obiettivi del programma e le modalità per raggiungerli, ero sicuro che la stesura del codice non avrebbe portato via molto tempo. Mi sbagliavo: già dalle prime versioni WPXDLL causava di tanto in tanto delle eccezioni in altri programmi, senza che riuscissi a capire perché, ad esempio, chiudendo WordPad si verificasse un "page fault" in WinOldAp (il processo che gestisce le finestre DOS). Il bug si è rivelato alla fine abbastanza insidioso, tanto che vale la pena di parlarne per evitare al lettore di trovarsi in situazioni analoghe. Compilando e lanciando WPX da una sessione DOS, alla chiusura di WordPad la finestra DOS veniva automaticamente riattivata, con conseguente inserimento di WPXDLL nel processo WinOldAp (a causa dell'hook WH_SHELL). Ciò è perfettamente normale e lecito in Windows 95, tranne per il fatto che, evidentemente, una DLL caricata in questo modo viene trattata in modo diverso dal sistema operativo. Dopo qualche prova infatti, la causa del "page fault" è risultata essere l'utilizzo dei registri fs, un segment register introdotto con le CPU di classe 386, da parte del codice di startup della DLL (in ambiente Borland C++ 4.5). Dato che esiguità e librerie dinamiche condividono più o meno lo stesso codice iniziale, secondo me il bug non è imputabile alle librerie fornite con il compilatore, che ormai ho usato con successo in diversi progetti, bensì al modo in cui vengono gestite le DLL contenenti funzioni di hook.

In ambiente Win32 i parametri per l'entry point di un programma o di una DLL vengono passati nello stack piuttosto che attraverso i registri della CPU, quindi è possibile scrivere in C puro anche il codice di startup. Il linker Microsoft permette infatti di impostare l'entry point con l'opzione -entry, che però manca nel linker Borland da me utilizzato nel progetto. Fortunatamente non è difficile riscrivere il modulo di startup per simulare la precedente opzione: esso consiste della singola istruzione

```
jmp DllEntryPoint
```

(il modulo deve essere scritto in Assembly per poter usare la direttiva "END [entrypoint]"). L'unico svantaggio è che la funzione di entrata della DLL deve chiamarsi necessariamente DllEntryPoint, ma ormai questo nome è abbastanza diffuso e standardizzato ed è una buona idea utilizzarlo in ogni caso. Un effetto collaterale da tenere presente quando si decide di non utilizzare il codice di startup fornito con il compilatore è che diverse funzioni di libreria dipendono per un corretto funzionamento da alcune operazioni di inizializzazione, le quali vengono per l'appunto eseguite al momento della partenza e prima che il controllo venga ceduto al codice scritto dall'utente. Nel dubbio, e se possibile, è meglio fare del tutto a meno delle funzioni di libreria del C o C++, affidandosi invece a quelle messe a disposizione dal sistema operativo. L'attuale versione di WPXDLL non usa librerie standard ed è più piccola del 60% rispetto alle precedenti versioni (che oltretutto non funzionavano a causa del problema descritto sopra), il che è particolarmente importante per una DLL che può essere potenzialmente inserita in quasi tutti i processi.

Il controllo RichEdit

Come si diceva in precedenza, la maggior parte del lavoro di WordPad viene svolta dal controllo RichEdit che si occupa sia della formattazione che della presentazione del testo e di eventuali oggetti OLE in esso presenti. Si tratta di un controllo molto potente, e anche se parlarne non è lo scopo principale di questo articolo (la documentazione "ufficiale" è molto curata e completa) è comunque utile conoscere almeno le principali differenze rispetto ai più conosciuti controlli Edit. La prima cosa che si nota è la divisione del testo in paragrafi, i quali possono essere formattati indipendentemente l'uno dall'altro con una grande varietà di opzioni che vanno dal tipo di giustificazione fino al supporto per il numbering, nella versione attuale limitato ai bullet. All'interno di un paragrafo si possono specificare caratteri con attributi diversi, ed anche se in questo caso il supporto è abbastanza limitato si tratta comunque di quel che basta per evitare di dover ricorrere tutto da capo nel caso si abbia bisogno di un controllo di testo con supporto OLE. Una interessante novità è il supporto di un protocollo stream-oriented per scrivere o leggere il testo del controllo: in questa modalità il controllo stesso ad allocare un buffer per il testo ed a chiamare una funzione specificata dal programma per compiere le opportune operazioni. Nel caso di WordPad, dove è necessario prendere il testo ad una certa finestra, è decisamente la più ... di questa funzione. Attenzione però perché nel RichEdit la dimensione massima del testo non è più limitata a 32K, ma può essere modificata con il messaggio EM_EXLIMITTEXT ed assumere valori molto superiori. In queste situazioni non è quindi consigliabile allocare un unico buffer per WM_GETTEXT, né del resto risulta particolarmente comodo allocare un buffer più piccolo e utilizzare ripetutamente WM_GETTEXTRANGE, specialmente quando è presente una selezione.

Ben vengano dunque gli stream, ma le novità non sono finite. I controlli RichEdit non si tirano indietro nemmeno quando si tratta di stampare, un'operazione che se effettuata con le API standard risulta sempre molto laboriosa. Con i messaggi EM_FORMATRANGE ed EM_SETARGDEVICE è possibile formattare il testo e copiare il risultato su un qualsiasi device context, e se quest'ultimo è associato ad una finestra piuttosto che ad una stampante nessun problema: con poco sforzo si ha un'anteprima della stampa finale.

Conclusione

Sarebbe un errore considerare WPX come un semplice esercizio di programmazione. Al contrario i metodi utilizzati possono essere impiegati per realizzare facilmente degli add-on su programmi che non supportano un'interfaccia pubblica verso l'esterno. Per fare un esempio in tema, sarebbe possibile estendere ulteriormente WordPad aggiungendovi una funzione di controllo ortografico. Naturalmente, questo articolo è stato scritto in WordPad/WPX con un rilassante sfondo grigio...

Nota: questo articolo è stato pubblicato sul numero 46 di *Computer Programming* (Aprile 1996).

I sorgenti si possono scaricare dal sito www.ascotti.org (http://www.ascotti.org/programming/zip/wpx.zip).

Copyright (c) 1995-2003 Alessandro Scotti.