CBT

Rosie Perera

ABSTRACT

This document describes everything there is to know about CBT in Word For Windows[1]. It is mostly applicable to other applications as well.

Contents

Introduction 2-1
Application Support for CBT 2-3
Initialization and Files 2-3
Termination and Cleanup 2-4
Window Hooks 2-6
Subclassing 2-6
Window Messages 2-6
Semantic Events 2-7
SDM 2-8
Other Coding Considerations 2-9
Appendix - Semantic Events 2-10

Introduction

Word For Windows uses Microsoft's standard Windows Computer-Based Training (CBT) system. Word For Windows's CBT consists of a tutorial. CBT can also include a feature guide, as in Excel, but Word For Windows doesn't have one. In this document I will use the terms "CBT" and "tutorial" interchangeably.

The Windows CBT system is an application which runs symbiotically with the application being taught. This means that the CBT communicates with the instructed application, both driving, or *ghosting*, the application by supplanting user-initiated events with CBT-supplied ones, and receiving events from the application indicating what user-initiated events have occurred. Communication between the instructed application and the CBT is achieved through the use of window messages, Windows hooks, and by subclassing the application's window procedures.

CBT bears a strong resemblance to a finite state automaton in that it is a collection of discrete states, including a unique starting state, and a well-defined set of transitions which are pairings of events and resultant states. The states in the CBT are referred to as *steps* and include definitions of

[1] In composing this document I borrowed heavily from a memo by CB Leyerle, "Windows Application Support for CBT".

what constitutes both the visual appearance and the true internal state of both the CBT and all applications present in the CBT session[2].

There are three types of events, user-initiated events, CBT-internal events, and semantic events. User-initiated events include keyboard input (both key-up and key-down), mouse button down-clicks and up-clicks, and mouse cursor movement. CBT-internal events include timers and configuration branching, and the application doesn't need to worry about these. The most interesting events, from the application's point of view, are *semantic events*, which will be described in detail below.

Tutorial lessons can include both non-interactive steps and interactive steps. In non-interactive steps, the lesson just explains how to do a particular task and the user sits back and watches while the CBT puts up *instructional windows (IWs)* and drives (ghosts) the application. In interactive steps, the user is instructed to do a particular action and the CBT branches to the next step only when the user does the correct action. It usually displays some sort of help message if the user does the wrong thing. These transitions from a step to the next step or to a help message are collectively called *Response Analysis (RA)*. The CBT help messages are called *Response Analysis Windows (RAWs)*.

The definition of what is the "correct action" for a user to do at a given step is made by the CBT *author* who creates the lesson. The author decides what he or she is trying to teach and what the user should do at this step and then records these events into the CBT lesson using the authoring tool DOT. This procedure is called *eventing*. Later, when the CBT is running, the events the user generates are compared against the pre-recorded ones and the CBT decides whether the user did the correct thing. The events may not need to coincide exactly, but in most cases they do. It is up to the author to event alternate methods into the lesson if the CBT should allow the user to do an action in one of two ways (for instance, choosing a command from a menu can be done with the arrow keys, or with the initial letter of the command).

Ghosting is simply a sequence of events which the application is to handle, and is usually sent at the beginning of every step in every lesson and appears as a normal sequence of incoming window messages to the application. Ghosting is used to demonstrate features (the app looks like it's in "autoplay" mode), as in most of the Word For Windows CBT lesson overviews. Ghosting is also used to modify the appearance and internal state of the instructed application without keyboard or mouse input, for example to close documents that were in use by a lesson when a new lesson is started, or to turn off the ribbon in preparation for teaching how to turn it on. The application need not to do anything special to handle ghosting when the CBT is running. Ghosted events are sent via the Windows journalling playback hook, so they appear as a sequence of regular window message to the application.

Semantic events are generated by the application from user-initiated events in the process of interpreting their meaning. These events encode more information about the impact of a user-initiated event than could be derived from the event alone without knowing the internal state of the application. The CBT needs semantic events only in those places where normal window messages and window books do not provide enough information. For example, selecting some text is an event which is comprised entirely of mouse and/or keyboard input, but which the CBT can't understand from that raw message data because it doesn't know where Word For Windows's text is displayed on the screen in mouse coordinates. So Word For Windows generates a semantic event for the CBT (by sending it a window message with cpFirst and cpLim packed into the lParam) telling it that text was selected with a particular range of cp's. Note that the CBT doesn't know that the numbers we send it are cp's, or even what cp's are, but all it needs is a unique way of identifying that event to make sure the user's action is identical to what the author evented for that step.

[2]Can be more than just the instructed application, as in the Help Lesson in Opus CBT.

In the case of user-initiated events, if the CBT decides that the user has not done the correct action, it will not allow the application to ever see the window messages associated with that user event. This is accomplished by the use of Windows hooks. This way the visual and internal state of the application will not change on an interactive step unless and until the user does the right thing. With semantic events, however, it is a cooperative effort between the application and the CBT that prevents the action from taking place if the user tries to do something that he was not supposed to do. The application basically has to ask permission of the CBT to allow a particular action to take place. If the CBT says yes, the application continues along as if CBT wasn't running. If the CBT says no, the application must abort the action. This will be described further below.

Application Support for CBT

Initialization and Files

Because CBT is entirely deterministic, it needs to proceed from a known state of the application. The authors decide what state they want to begin from and all of their eventing and ghosting is based on this assumption. Things that define that "starting state" for Word For Windows include having no documents open, having the ribbon and ruler off and the status line on, etc. See FInitDefaultPrefs and FInitStateForCBT for how these are all set up. It is extremely important that the application start in the same state that the author had it in when authoring a lesson. Otherwise the lesson won't work. This means that if new items are added in future versions which are part of Word For Windows's global state (especially new fields in vpref), they should be set properly for CBT and that information should be communicated to the CBT team.

After the CBT is terminated, the state should be restored to exactly what it was before the Tutorial was run. Thus we save away the current state before setting up the CBT starting state. The saving and restoring of global state in Word For Windows is a pretty quick-and-dirty one. If there's time in a future version, we should make it more complete. There are lots of items that are not restored, such as whether a window was in outline mode, whether there was a split pane, header/footer panes, open macro windows, etc.

The CBT consists of the executable file (winword.cbt), the CBT dynamic library (winword.lib), the lesson file (winword.les), and the various templates and documents that the CBT has Word For Windows open during the course of the lessons. These files all reside in a special CBT subdirectory (winword.cbt[3]). You may see the filenames wincbt.exe, wincbt.lib, and wincbt.les when working with the UserEd team. These are the generic names of the above files, respectively. They are renamed for specific projects. Actually, wincbt.lib (winword.cbt) is the only one which is not project specific. A special wincbt.exe is built that is Word For Windows-aware. And of course our wincbt.les (winword.les) is entirely Word For Windows-specific.

During the time that the CBT is running, the DOS current directory must be the CBT subdirectory, in order for all the documents and templates to be found. So one of the first things Word For Windows does when the user chooses Help Tutorial is to switch into the winword.cbt directory. The old current directory is saved so it can be restored later. Word For Windows finds the winword.cbt subdirectory by looking first in the util-path directory (if a util-path exists) and then in the program directory (the one where winword.exe is).

The last thing Word For Windows does before booting CBT is shrink the swap area size. This will enable the CBT to have enough memory to boot. After CBT is successfully booted, we grow the

[3]Yes, the directory has the same name as one of the files in it. I tried to fight this but lost.

swap area size back to maximum. This is an area that could be tweaked somewhat if it turns out that CBT runs out of memory too often during lessons (CBT is very data intensive). We could leave the swap area size down[4], or grow it back only part way. I believe this was discussed a little before we shipped but we decided it was too late to monkey with it. I don't know of any serious problems with it.

Word For Windows launches the CBT by first loading the dynamic library, and then booting the CBT executable. If either of these files (or the winword.cbt directory) is not found, Word For Windows cannot run the tutorial. The CBT does some checks to see if there is enough memory for it and to see if all the files it needs (winword.les and all the specific document and template files required by the lessons) are present, and will terminate itself if not. Once we have booted the CBT executable, we wait for it to send us a WM_CBTINIT message, indicating that it has successfully booted. If instead we get a WM_SYSTEMERROR, something went wrong. We use the global tri-state flag vrf.fCbtInit to communicate between AppWndProcRare and FRunCbt so that we know when CBT is up and running.

Once CBT is booted, the application needs to send CBT a WM_CBTINIT message with its instance handle (so it knows who booted it) and then WM_CBTNEWWND messages with the handles of all its currently open windows. Certain child windows can be omitted from this enumeration if they never handle mouse or keyboard input in their window procedures (e.g. in Word For Windows, the split bar is a window which does not take input). The reason for sending the window handles is so that the CBT can subclass the windows (this is described below).

With all of this done, the application is now in CBT mode, and CBT is in control. The application can determine whether CBT is active or not by the value of hwndCBT (vhwndCBT in Word For Windows), provided this is initialized to NULL at application boot time, and set/cleared as appropriate according to receipt of WM_CBTINIT and WM_CBTTERM messages.

Termination and Cleanup

Termination of the CBT can be done at the instigation of either the CBT or the application. The CBT will seek to terminate either when the user requests it, or when a fatal condition (such as out-of-memory) is detected. The application will seek to terminate the CBT when it detects an error condition such as OOM. After the CBT terminates, the application can proceed as normal. If an error condition is not deterministic given the user-initiated events and semantic events, the CBT won't be expecting the error message and will not allow the user to OK the message box. Sometimes this requires terminating the CBT, otherwise a semantic event (sent *before* the message box goes up) could solve the problem. The only non-deterministic condition which causes Word For Windows to have to terminate CBT for a reason other than OOM is if we try to open a document which is of an old file format. If one of the documents or templates necessary for a lesson cannot be opened, obviously the lesson cannot continue. I did review *all* of the Word For Windows error message to see which of them were deterministic and which weren't, and it is sometimes a very difficult thing to determine this. There may be some weird cases I've missed.

When the CBT wishes to terminate, it will broadcast a WM_CBTTERM message to all windows. The application should have one window procedure (for a window that won't get destroyed during the CBT, generally the main window), which processes this message. Word For Windows handles this in AppWndProcRare. It is important to remember that the termination can occur at any time, and in particular may occur with a menu dropped down or a dialog box or message box displayed, or while the application is in any temporary mode. Thus, the application needs to do a general cleanup which does at least three things: end any active menu state (via the undocumented Windows call

[4] From a comment in CB's memo, it seems that this is how Excel does it, though we copied our code from them so we must have both done it the same way at that point.

EndMenu[5]), close any open dialog of the application, and close all open documents, workspaces, etc. without prompting or saving. Word For Windows has some special concerns here, namely drop-down list boxes from icon bars, which are closed using TermCurIBDlg.

If the application desires to terminate CBT, it sends a WM_CBTTERM message to the CBT window. The wParam must be the instance handle of the application. For an error condition, the low word of the lParam of this message must be zero; the high word may contain a specific error code if desired. If the application is quitting normally, i.e. without any error condition being detected, and is in CBT mode, the low word of the lParam should be non-zero[6]. The CBT will broadcast a WM_CBTTERM upon receipt of the message from the application, and the termination sequence continues per the description above.

In an error situation, the application *must* send the WM_CBTTERM message *before* displaying a message box. Failure to do this can lead to a deadlock condition, and necessitate a reboot. Also, we discovered that sending CBT more than one WM_CBTTERM message can be harmful, so if there can be nested error conditions or if cleanup from one error condition can cause another one, it is important for the application to note that it's already sent a WM_CBTTERM to that CBT doesn't have problems. Word For Windows uses vmerr.fSentCBTMemErr to keep track of this.

Currently CBT does not put up any message when it is terminating due to user request or application request. It is up to the application to put up an error message (if it wants one) if the CBT is being terminated due to out-of-memory or other error condition. There was some talk between Raman (suryanr) and myself about having CBT having some smarts about the error code in the high word of the lParam, but currently that code is never checked. In some cases, it may be useful to use semantic events for certain error conditions so that authors can decide how to handle the problems. For instance, in the Using Help lesson, if Word For Windows cannot find the Help application to run it, we send CBT a semantic event message about this, and then the lesson takes a different branch which tells about what Help *would* be like if the user could run it.

One thing which needs to be changed for Winword 1.1 and PM Word, which will save a tremendous amount of space in the lesson file, is the way lessons do their own cleanup. When the tutorial terminates, the application does certain cleanup items, as mentioned above. Currently, if a user elects to restart a lesson or go to a different lesson either at the end of one lesson or in the middle of one, the lesson is responsible for remembering what has changed since the start state and resetting all those things. It uses internal variables to do all this. The authors have to set them properly and figure out the cleanup. Unfortunately this is a big waste of space because a lot of it is duplicated in many lessons. Also, it is silly to have the lessons do all this cleanup work when Word For Windows already knows how to do it. So what has been proposed is to have a lesson ghost "Help Tutorial" when it wants to clean up, and have Word For Windows know that if CmdHelpTutorial is called when CBT is already active, it means we should do the cleanup. In order for this to be accomplished, the code which sets up the initial CBT state will have to be isolated into its own routine. Saving the state away should only be done once, and FInitStateForCBT should be made so it can be called multiple times in a single CBT session. This should not be too difficult to do.

After CBT is terminated, the application should free the CBT library, using the windows call FreeLibrary. We had some fun with this one in Word For Windows. This library shouldn't be freed until it is very certain that CBTLIB is no longer on the stack, otherwise certain death will occur. Word

[5]This call has been removed in Win 3.0 and we must now use the WM_CANCELMODE message, which is documented.

[6]In Opus, we never terminate CBT except for an error condition. I can't imagine a tutorial ever teaching the user to quit the application and allowing it to actually quit, so I'm not sure why this case is

For Windows now postpones this and the rest of the post CBT cleanup (StopCBT) until after vfDeactByOtherApp is false.

Window Hooks

CBT uses several Windows hooks, including the keyboard hook (WH_KEYBOARD), the journalling hooks (WH_JOURNALRECORD and WH_JOURNALPLAYBACK), the system message filter hook (WH_SYSMSGFILTER), and the CBT hook (WH_CBT). The journalling hooks are used to record and ghost, respectively, the keystroke/mouse-event sequences needed to keep the application in the right state for the CBT instruction. The others are used to intercept user-initiated events so that they can be compared to the authored RA list (events which the author decided would satisfy the instructions and thus cause the CBT to branch to the next step). The application can use these hooks as well, and, except for the journalling hooks, almost without restriction. If the application intends to use the journalling hooks, other arrangements will need to be made, and in any case, applications development and CBT development should mutually ensure that their respective use of hooks is orthogonal.

Subclassing

When the application creates a new window, it must, when CBT is active, tell the CBT about the window handle so that the window can be subclassed. This is accomplished by sending a WM_CBTNEWWND to the CBT window, with wParam as the hwnd of the new window and lParam as 0L. This is normally done upon receipt of the WM_CREATE message in the new window's window procedure.

Subclassing is a technique whereby the CBT interpolates its own window procedure between Window's dispatch and the applications window procedure. This produces a chain of window procedures with the CBT subclassing window at the head of the chain. The CBT interposes its own window procedure upon receipt of the WM_CBTNEWWND message, and removes it when it sees a WM_DESTROY for that window as a result of its subclassing. Subclassing allows the CBT the opportunity to examine the various messages destined for the window, and intercept those that would, for CBT purposes, incorrectly modify the application state. Most messages are, after examination, immediately sent to the application via CallWindowProc.

Window Messages

The application needs to define some special CBT window messages, which are reserved in windows.h, but not specified therein. These are WM_CBTINIT (0x3f0), WM_CBTTERM (0x3f1), WM_CBTNEWWND (0x3f2), and WM_CBTSEMEV (0x3f3). The first three were explained above, and the fourth will be explained below. An additional message which may be needed by some applications was defined to solve a problem in Whimper and Word For Windows CBT's. This is WM_CBTWNDID (0x3f4). It is used to give the CBT a unique id for a window to allow it to distinguish between multiple windows of the same class when it (the CBT) doesn't know what order they were created in. For instance, in Word For Windows, the problem arose with icon bars[7]. The window handle cannot be used, because it will be different each time Word For Windows is run.

---

specified. But that's what CB's memo says. I looked in the CBT code, and discovered that unless the low word of lParam is zero, CBT doesn't actually terminate.

[7]You may ask the obvious question, "Why not send the id # in the (otherwise unused) lParam of the WM_CBTNEWWND message? Well, I asked that too, and was told that it has to be this way because

Semantic Events

Semantic events were introduced briefly above. As promised, I will go into more detail here. There are two kinds of semantic events, advisory and pre-emptive (abortable). Advisory semantic events are when the application just notifies the CBT of some event which has already occurred. These include things like selecting text in Word For Windows, where the lesson allows the user a certain amount of exploration and doesn't prevent actions. The lesson will branch to the next step only when the event it is looking for has occurred (the user correctly selects a particular word, for instance).

Abortable semantic events are when the application detects that the user is trying to do something but must prevent it if the CBT says it isn't the right thing to do at this stage. To provide CBT with this power, the application breaks the handling of this type of action into the two phases of *recognition* and *doing*. The recognition is the determination that a mouse click at a particular coordinate position means the user is trying to choose Bold from the ribbon, for instance. The doing involves carrying out the Bold command. The CBT needs to know that the application intends to carry out the Bold command before it actually is done. Thus, Word For Windows sends the CBT a semantic event between these two phases, to indicate its intent. If the CBT responds (by returning fFalse from the WM_CBTSEMEV message handling) that it does not want the user to do this action at this time, Word For Windows must abort out of the action. We have to be careful not to leave things in a half-done state. We must clean up the screen or whatever, particularly in cases where the user was trying to drag something where they weren't supposed to.

In most cases, the distinction between recognition and doing is not conscious, but there were times when retrofitting CBT hooks into Word For Windows where we discovered that code had been written in such a fashion as to make it difficult to install an abortable event, so the CBT authors had to live with an advisory one and work around it (the abortable ones are more desirable for them instructionally, but they can be more work for the application). Here is the skeleton code for a semantic event:

```
if (vhwndCBT && !SendMessage(vhwndCBT, WM_CBTSEMEV, wType, lData))
        {
        /* CBT vetoes attempted user action.  Abort it. */
        goto LCleanup;
        }
else
        {
        /* CBT not active or event OK.  Do normal handling */
        }
```

The semantic event type is sent in the wParam. These are ID numbers defined by the application and communicated to the CBT developers (because the CBT executable needs to have them built in). The lParam may contain additional data for the semantic event if necessary. All of the Word For Windows semantic events are listed at the end of this document (or chapter, if you're reading the whole Word For Windows Fundamentals document).

Most semantic events are the result of mouse actions, because keyboard messages tend to contain all the necessary information for the CBT to tell what the user is doing. One exception is selection via the cursor keys. There could be an infinite number of ways of moving the cursor to one point in the document. The CBT doesn't care what actual keystrokes the user took to get there, but just that they get to the right place. This warrants a semantic event.

WM_CBTWNDID was invented as an after-thought and it was harder for Whimper to go back and move all their WM_CBTNEWWND calls (apparently they didn't know the id # yet at the time of the WM_CBTNEWWND call) than to add a second message later on. So we do it their way.

For mouse semantic events it is necessary to decide whether to send the semantic event on the down-click or the up-click. It depends on the nature of the action and the needs of the CBT authors. For example, in Word For Windows clicking on an icon in an icon bar generates a semantic event with the ID of the icon clicked. We don't actually carry out the action until the mouse button is released (though we do highlight the button so the user knows they clicked on it). In this case it is better to send the semantic event on the up-click. It won't prevent the highlighting from occurring if the user clicks in the wrong place, but the lesson should not branch until the user actually does the thing that will cause the action. Another example is dragging things in outline mode. Again, here, we don't take action until the up-click, though we do animate the dragging. We send the semantic event on the up-click. Here we have to be careful when we abort the command that we don't skip the code that cleans up the screen (turns off the animated drag outlines and the bullet highlight).

Generally I would recommend designing commands that take action on mouse clicks such that they take their action on the up-click. This is in keeping with standard practice in Windows apps and makes it easier to do the CBT hooks. But unfortunately we have a counter-example where the user clicks the arrow to drop down a combo box from an icon bar. In this case, we drop the list box down on the down-click, so we need to send the semantic event on the down-click. Actually, this particular example is one of SDM's semantic events, which will be discussed below.

The specification of semantic events requires the close cooperation of the application developers, CBT development, and the CBT authors. The authors decide what semantic events they need to instruct certain aspects of the application, and the application developers decide how to convey the necessary information in three words and put in the code to do so. Then the CBT developers put in support for the semantic events as specified, and the authors do the eventing for them. I advocate that this specification should be done as early as possible so that the code can be designed to allow aborting easily, but it is usually not difficult to put semantic events in later on if another need arises.

The application developers should be aware of what kinds of things might need semantic events. In Word For Windows, we ended up having to add a lot of semantic events later on for things that the CBT group didn't think of. In particular, one thing we realized was that even though a particular mouse action is not going to be taught in any lesson, you may well need a semantic event for it, to *prevent* people from doing that action when they're supposed to be doing something else. The prime example was dragging the style name area bar. This will be necessary any time a mouse command is generally available when something else is being taught. If it is not available anyway, no semantic event is needed.

SDM

When SDM has control (i.e. when a dialog box is up), Word For Windows doesn't know what events are happening, but the CBT still needs to know so that it can teach users how to use dialog boxes. Therefore, SDM has CBT hooks in it too. In order to make SDM aware that CBT is present, we use the SDM call CBTState, passing fTrue. Similarly, when CBT goes away, we call it again with fFalse.

SDM sends the CBT semantic events for clicking on items in dialogs, typing in edit controls, selecting in list boxes, etc. As mentioned above, CBT is responsible for the semantic event of clicking to drop down an icon bar list box. There needs to be tight communication between the SDM developers, CBT developers, Word For Windows developers, and authors, in order for all of this to work smoothly.

Other Coding Considerations

Applications with CBT support should not process messages returned as a result of calling PeekMessage with PM_NOREMOVE. Applications may use PM_NOREMOVE to see what (whether) messages are available, but should not process them until they have been removed from the queue.

The WM_QUEUESYNC message *must* get through to CBT, because it uses that to manage its journalling playback hook. We have a place where we need to do a PeekMessage with PM_REMOVE in a tight loop. PM_REMOVE is normally OK, but if a WM_QUEUESYNC comes along, it must be passed on to CBT or you'll be in an infinite loop. We do not need WM_QUEUESYNC cases in our window procedures because CBT gets this message via its subclassing.

If a key is defined to abort operations in progress (such as printing, repaginating, etc.) it should be disabled when the CBT is active. Word For Windows does this in FCheckAbortKey.

Appendix - Semantic Events

Here is a list of all the semantic events in Word For Windows 1.0 and their parameters. This list will surely become outdated in future versions of the CBT. These are not documented with their parameters anywhere else, except by reading the code. Under "Type" I have encoded two letters; the first tells whether the event is advisory (A) or pre-emptive/abortable (P), and the second tells (where applicable) whether the event is sent on the down-click (D) or the up-click (U).

Semantic Event	ID	Type	HIWORD(lParam)	LOWORD(lParam)	Description
smvSelection[8]	0x0200	AU	cpLim	cpFirst	/* new text selection */
smvHdrSelection[9]	0x0201	AU	cpLim	cpFirst	/* new header selection */
smvFtrSelection	0x0202	AU	cpLim	cpFirst	/* new footer selection */
smvFNSelection	0x0203	AU	cpLim	cpFirst	/* new footnote selection */
smvAnnSelection	0x0204	AU	cpLim	cpFirst	/* new annotation selection */
smvAnnFNMark	0x0210	P	0 - footnote - annotation	cp	/* dbl click on Annot or FN mark */
smvIBHdrFtr[10]	0x0220	PU	button group	button within group	/* header/footer iconbar buttons */
smvIBOutline	0x0221	PU	button group	button within group	/* outline iconbar buttons */
smvIBMcrEdit	0x0222	PU	button group	button within group	/* Macro edit iconbar buttons */
smvIBRibbon	0x0223	PU	button group	button within group	/* ribbon iconbar buttons */
smvIBRuler	0x0224	PU	button group	button within group	/* ruler buttons */
smvIBPrvw	0x0225	PU	button group	button within group	/* print preview iconbar buttons */
smvTabCreate[11]	0x0230	PU	x pos (in xa's)	0	/* create new tab on ruler */
smvTabMove	0x0231	PU	old x pos	new x pos	/* move an existing tab */
smvTabDelete	0x0232	PU	0	x pos	/* drag tab off of ruler */
smvIndentBoth	0x0233	PU	old x pos	new x pos	/* drag both left indents together */
smvIndentFirst	0x0234	PU	old x pos	new x pos	/* drag first line left indent */
smvIndentLeft	0x0235	PU	old x pos	new x pos	/* drag left indent */
smvIndentRight	0x0236	PU	old x pos	new x pos	/* drag right indent */
smvRulerTLeft	0x0238	PU	old x pos	new x pos	/* drag left table col mrk on ruler */
smvRulerTableCol	0x0239	PU	old x pos	new x pos	/* drag table col 'T' mark */
smvMarginLeft	0x023a	PU	old x pos	new x pos	/* drag left page margin */
smvMarginRight	0x023b	PU	old x pos	new x pos	/* drag right page margin */

[8]We pack two cp's into one lParam because that's all the room we have. This means the CBT authors have to limit their documents to 64K. Not a very tough restriction.

[9]The authors wanted a separate semantic event for selection in different panes since the cp spaces in other panes overlap and they need to be able to distinguish the event. All selection events are sent both with mouse selection and with keyboard selection.

[10]The "button group" is an index, starting at 0, of the group of buttons (as they are visually laid out on the iconbar). The button within group is the index of the button within the group. See code for specific id numbers.

[11]All the ruler semantic events (0x0230-0x023b) are sent on the up-click, so we need to clean up properly and remove the tab mark or put ruler marker back where it was if the semantic event is aborted.

Semantic Event	ID	Type	HIWORD(lParam)	LOWORD(lParam)	Description
smvPrvwDrag	0x0240	AU	0 - no shift key 1 - Shift key	index of object: 0 - left margin - top margin - right margin - bottom margin - header - footer - page break on up_ APO's	/* Prvw: drag margins, etc */
smvPrvwUpdate	0x0241	AD	<unused>	<unused>	/* update print preview display */
smvPrvwOtherPage	0x0242	AD	<unused>	<unused>	/* Click on other page */
smvPrvwPageView	0x0243	A	<unused>	<unused>	/* doubleclick to enter page view */
smvPrvwClick	0x0244	AD	<unused>	<unused>	/* click when borders not showing */
smvTableSelection[12]	0x0250	AU	cpFirst	MS to LS nibble: Row Min Row Mac Col Min Col Mac	/* select cells in a table */
smvPicFormat	0x0260	AU	ircs (handle id)	0 - cropping - scaling	/* crop or scale a picture */
smvPicSelImport	0x0261	AU	0	ifld (index of field)	/* select an import pic (tiff) */
smvNonPicSel[13]	0x0262	AD	<unused>	<unused>	/* Sh+click outside of sel'd pic */
smvOutlineSelect	0x0270	AD	0	cpFirst of paragraph	/* Click outln icon to select para */
smvOutlinePromote	0x0271	PU	lvl promoted to	cpFirst of paragraph	/* Promote/demote paragraph */
smvOutlineMove	0x0272	PU	destination cp	cpFirst of paragraph	/* Move para up or down */
smvOutlineExpand	0x0273	P	0	cpFirst of paragraph	/* expand/collapse text (dbl clk) */
smvBeginTyping[14]	0x0280	P	<unused>	<unused>	/* entering the insert loop */
smvCommand[15]	0x0290	P	0 - from menu - from keybd	id (== help context)	/* Command event */
smvTrackStyWnd	0x02A0	PD	0	0	/* Click to drag style area */
smvCantBootHelp[16]	0x02FE	P	0	0	/* OOM or can't find help */
smvWmChar[17]	0x02FF	P	0	VK (virtual key code)	/* WM_CHAR received */

[12]Since the row and column Min and Mac must all be squeezed into a long, we use four bits for each, so the CBT authors need to restrict any tables in their sample documents to 16x16.

[13]A picture is selected and the user tries to extend (using Shift+click) the selection to somewhere outside the picture. This was added as a preventative semantic event to fix a bug in a lesson.

[14]The need for this event was due to a "feature" in the design of DOT, whereby the authors can not prevent typing text while still allowing other keyboard input without authoring an event for each possible key that would cause Opus to enter the insert loop.

[15]Sent when we receive a WM_COMMAND or when any command is run from a key. The CBT needs this because Opus uses bcm's for menu id's, and these can change from one build to the next. CBT, like Help, needs a constant id to be sent, so we just use the ibcm's which are used for Help.

[16]This one is *very* abortable. If CBT vetoes this semantic event, we terminate CBT.

[17]This was needed because Opus does PeekMessages in our insert loop, so CBT doesn't get the WM_CHAR messages. CBT has the keyboard hook installed which picks up most keys, but if the user does an Alt-Numpad combination CBT depends on the WM_CHAR message, so we need to send them. This is particularly important for international versions. Internally, CBT treats this semantic event just like a WM_CHAR message.

Command Dispatch

Bradford Christian

ABSTRACT

Command dispatching in Word For Windows is done from a central, generalized function that provides command name overloading and treats built-in commands and user-created macros exactly the same. This document describes the command dispatcher used in Word For Windows.

Contents

Introduction 3-1
The Command Table 3-1
Command Functions 3-2

Introduction

A familiarity with SDM is assumed.

The Command Table

Commands

Every command has an entry in the command table called an SY. An SY contains common information about a command such as it's name, type (dialog, EL, other, etc.), a pointer to it's function or macro and flags that indicate when the command may or may not be used.

MkCmd

The MkCmd tool reads the .CMD files and turns them into code space structures that are copied to the command table during initialization. Also produced are a resource file containing the non-modifiable menus and a header file that defines the bcm values.