The CWIN-2 Windows Interface Code

A C Norman

October 19, 1995

1 Introduction

This document is not too concerned about internal data structures or programming issues, but will attempt to document the behaviour I will implement and in some cases explain why I did things that way. The explanation is needed because while writing the code (and its predecessors) I found it hard to make scrolling, selection, cut & paste operations, font changes, user input and program output all interact properly.

First the fundamentals. I will support a window that always has both horizontal and vertical scroll bars visible.

To start with it will contain just text, but parts of that will be shown in differing colours to mark out where there are prompts. I will allow for the possibility that in a later version the text in the buffer can be displayed in various fonts, sizes and with various effects. The unit of display will be a "line" which is represented as a block of characters and all effects withing the block will be indicated by embedding control sequences within it. Each line will start off in a standard font configuration.

The user types in characters, uses the DELETE key, activates scroll bars, uses the mouse to re-position the caret and/or establish a selection region, and invoked CUT and PASTE operations.

There is a program running that will, from time to time, request characters from the user. Before doing so it can select the prompt that should be used. It can also generate output. There is no automatic supposition about how long the program delays between I/O requests and how it interleaves reads and writes.

The stored text lives in a buffer that has finite size, and so after a program has done enough the buffer will fill up and parts of it will need to need to be recycled.

2 Scrolling under user control

If the whole of the text buffer can fit on within the window vertically the the vertical scroll bar has no effect, and any attempt to position it to anywhere other than the "top" position has no effect. If there are more lines of text than will fit on the screen at once the vertical scroll bar (and related short-cut keys, viz page-up, page-down, home and end) cause the window to scroll. Scrolling is limited so that at one extreme the first line of input text is visible at the top of the window and at the other extreme the final line of text is just fully visible at the bottom of the window.

If the collection of lines displayed in the window are all short enough to fit completely across it then the horizontal scroll position is forced to its home (leftmost) position and attempts to move it have no effect. Otherwise the user may scroll horizontally up to an extreme where the longest line that lies within the window comes just half way across the window.

If the window size of font is changed scrolling can be activated to keep the conditions described above true. In particular this means that when the window is enlarged it will scroll to keep the last line of the text visible as the last line in the window, and to keep horizontally scrolled information visible at least half way across the window. Resizing the window will update the position the the scroll-bar thumbs to reflect what has happened. There can also be scrolling caused by input and output operations, but that is discussed in a moment under the heading AUTO- SCROLLING.

The END operation moves the caret to an extreme end of the text and scroll the window to make it visible. HOME scrolls the window to show the very top of the text, but does not re-position the caret.

3 Use of the mouse to select regions of text

The caret is considered to lie between a pair of characters (or right at the very start or very end of the text). Text forming a "prompt" is treated as representing a single item, and so the mouse can not select a position within the prompt text. Pressing the left mouse button causes the caret to be moved to the gap nearest to the mouse position. The place where the mouse button was depressed becomes the root of a possible selection region. From when the mouse button was depressed to when it is released the mouse is captured by the window, in the sense that even if the mouse moves outside the window's borders its activity is still tracked. If the mouse is moved with the button down a region of text becomes selected. The text in that region is displayed in a form of inverse video (exact details depend on the windows colour scheme in force). If the mouse is dragged above or below the window while making a selection the window is scrolled in the relevant direction, and the speed of scrolling increases if the mouse remains outside the window for a while. If the mouse moves to the left of the window and the window has already been scrolled horizontally it will be gradually restored. If the mouse goes to the right of the window scrolling will be activated if the line that the mouse is on extends beyond the right hand border of the current window.

Pressing the (left) mouse button with SHIFT held down results in a selection being extended from its original root to the new mouse position. Thus clicking in one place and then shift-clicking in another sets up a selection from the first to second place.

The shift-click operation relies upon a previously set selection root. If none has been established before then shift-click behaves just like an ordinary click. Once set by an initial click a selection root remains valid until another mouse click (which re-positions it) or until buffer wrapping deletes some characters from the buffer (eventually almost any input or output operation will cause this) or until the user deletes a character from the text using the DELETE key. CUT operations can leave a selection root set at the place where the removed material used to be.

The COPY operation copies all characters from the selection to the clipboard. If control characters are present they are sent to the clipboard without adjustment. Lines sent to the clipboard have CR/LF at their end as termination. CUT acts as COPY but then the selected region is deleted. RE-INPUT performs a COPY and then PASTEs the material into the input buffer(but because it uses COPY it places the selected text into the clipboard on the way). SELECT-ALL does what it says (and does not move the caret). CLEAR throws away all text in the display buffer. Immediately after a CUT operation the UNDO button will re-position the caret to the place where material was deleted and will perform a PASTE. Only one level of CUT can be UNDOne. A DELETE when a selection has been set up will delete the selection (but not do the COPY operation that a CUT would). Part of an effect will be that a DELETE may not keep quite so much stuff for UNDO as would a CUT.

COPY operations can fail if the clipboard can not be opened or if it proves impossible to allocate windows global memory for the text that was to be copied. In such cases the fact that the COPY failed will not be indicated to the user, and in the case of a CUT it will not be possible to UNDO the CUT.

4 Program-generated output

This is always inserted at the end of the buffer. If the caret is at the end of the text it moves as new text is inserted, and in this case if the caret starts off visible the screen is scrolled automatically to keep the caret position visible. This may involve horizontal as well as vertical scrolling, but an attempt will be made to delay auto-scrolling so that it does not happen too often.

Action has to be taken when the buffer becomes full. The buffer can be "full" either because the maximum number of characters have been stored in it or because it holds the maximum number of lines that it is configured for (at present I will allow for 64K characters and 2K lines). In normal circumstances when the buffer overflows the oldest whole line of text stored in it is discarded. This policy is modified in various circumstances:

1. If the buffer contains only one (incomplete) line of text then the system throws the entire buffer contents away. If there were unbalanced control sequences in the buffer the subsequent display may be mangled, for instance by being in the wrong colour or font. This case can only arise if the program being run keeps printing large numbers of characters without an intervening newline. I really hope this circumstance does not arise! I specify this extreme behaviour because it is hard to indicate a safe way of abandoning a part-line if I intend in due course to put elaborately interpreted in the buffer. A possible safer refinement here would be to discard any further characters inserted into the buffer until a newline. But at present it does not seem worth while legislating for such a desparate and (I hope) unusual situation.

2. If the line that is to be discarded overlaps with a region of text that has been selected the system pauses, changing the title bar of the window to alert the user to what has happened. This delay is so that the user can complete a COPY operation to preserve line that is about to be lost, or a PRINT (for the same reason). When the selection is cancelled for any reason the line will become vulnerable and output can continue.

3. If the line for that is to be deleted is not involved in a selection but it is visible on the screen and the end of the text (ie the insertion point) is not visible, then again the system pauses. This allows for the case where a user has scrolled up the display to inspect an early part of a transaction, and would not like it to vanish untill the scroll back down to the end. I allow the line to be deleted anyway in the case that all the lines of input are visible on the screen: this case can only arise when at least one of the lines of output is amazingly long. This is because the user could not then scroll the window to give the system a hint that it should proceed.

If output is generated and after it has been put on the screen the window is not scrolled down as far as it will go then output will need to update the vertical scroll thumb's position. For instance if before the output was generated the scrollbar thumb was at the 50% position, afterwards it will typically be higher up. Discarding old text can also influence the position where the thumb should be shown.

5 Keyboard and PASTE input

DELETE deletes the character before the caret, or if that is the end of a prompt it deletes the whole prompt. After a consecutive series of DELETE operations an UNDO will re-insert the deleted character, but there will be a limit to the number of characters that will be stored, so this can only be relied upon for small operations.

The effect of keyboard input depends on whether the caret is at the end of the text or within the buffer. To a more minor extent it also depends on whether the caret (while within the text as a whole) identifies a position within a currently active input line.

When the caret is not at the end of the text, normal characters are inserted before the caret. The region covered by an insert is recorded so that after a PASTE operation UNDO will discard what was inserted, while after individually typed characters each UNDO will delete either a block of characters typed without repositioning the caret or a block of characters that end at a newline. As usual there will be a limit to how much undoing can be done, and the exact rule for where that limit is is not documented and nobody should ever rely on it.

Inserting characters into the middle of the text like this might overfill the buffer, and in that case the first line of the buffer gets discarded to make space. There will be no delay in this even if the first line forms part of a selection (the starting point of the selection will move to the start of the next line, and maybe the selection will vanish totally). There will be a special case if the caret is in the first line when the user tries to insert more - in this case the insertion will fail. If it is from the keyboard the system beeps and discards the single offending character. If it is part of a PASTE then characters are ignored up to the next newline in the pasted text, which is notionally inserted and then the resulting first line is then thrown away. I guess in this case it means that if the caret is right at the start of the buffer all the inserted stuff gets abandoned.

Characters inserted at (or in some cases near) the end of the text can be used as program input. If the caret is at the end of the text typed-in chaarcters are placed in a type-ahead buffer until the program requests a line of input. When that happens characters are accepted from the type-ahead buffer (and/or the PASTE source) and echoed to the screen until a newline is seen. If the user types a newline at the end of the input line the characters in it are moved to a program-input-buffer which is where the program reads them from. If the user re-positions the caret and inserts a newline into the middle of the input line then pre-typed characters in the line but after where the newline was get pushed back into the type-ahead buffer (and if that overflows they are lost with a beep). The effect is that the program gets one line at once and when that line is placed in its input buffer it will just have been echoed to the screen. The program-input-buffer will have limited length and truly ridiculously long input will be silently truncated when moved into it. I will feel entitled to reject input activity that I notice creating an input line that is longer then that limit.

An elaboration on this explanation is that part of the final line in the text buffer can be an incomplete input line. This can start part way along the line (eg it will tend to start after the displayed prompt). After various CUT and DELETE operations or when the program requests input after printing a line that was not terminated it can start well along the final line. An incomplete line is created when the program requests a line of input. When the user inserts a newline into the incomplete line it becomes complete, its contents are moved elsewhere and there is no longer an incomplete line.

When a PASTE operation copies material into the middle of a document any prompts are inserted. But if then some of that line is moved out to the program-input-buffer prompts are discarded during the move. If PASTE puts stuff right at the end of the buffer it omits any prompts in the pasted stuff. But the start of each line of input that is echoed will get a fresh prompt displayed on it.

When the program that is being run is halted waiting for input and the screen has been scrolled such that the end of the buffer the window title is changed to "waiting for input".

Pressing any key or performing a PASTE operation always scrolls the window to make the caret visible. The caret can only have become invisible as a result of a user-initiated scroll request (or HOME) since except when such a request has hidden it the window scrolls automatically to keep it visible.

Note that the rules given here indicate that characters are only inserted into the buffer at two distinct places: where the caret is and at the end of the buffer. So the implementation can survive if it just caches information about those two positions.

6 UNDO — a summary

There is an undo buffer that can store a limited number of characters and a limited number of transactions. A transaction identifies a caret position or a range within the text, an possibly a sequence of associated characters:

1. After a PASTE that happened within the body of the text and did not terminate an input line an UNDO discards the inserted material;

2. After a PASTE that put one or more newlines into the input area no UNDO will be possible (because some of the inserted text has been passed on to the appication code to process);

3. After a CUT followed possibly by operations that move the caret an UNDO re-positions the caret and does inserts characters as for a PASTE (but that paste is not itself undoable);

4. After a sequence of DELETE keys have been pressed an UNDO will re-insert the deleted characters. It can re-instate deleted prompts.

5. Sequences of non-delete characters are collected up to the point of a newline. If the newline causes transmission of the characters to the program no UNDO is possible. Otherwise each block up to a newline is an UNDO unit.

6. Previously stored UNDO operations can become invalid if they overlap with a non-undo-able operation or if the text that they relate to gets abandoned as the main text buffer rolls, or if the undo stack becomes over-full. [Is this hard to implement reliably?]

7 Auto-scrolling — a summary

If the user never re-positions the caret it will remain at the extreme end of the text buffer. And scrolling will occus to keep it visible because

1. Program output will occurs with the caret at the end and visible;

2. Input requests will be handled with the caret at the end of the buffer.

If the user scrolls the window back (using the scroll bars) so that the caret becomes invisible then scrolling on output will cease, but the screen will jump to make the insertion point visible if a key is pressed and the program is requesting input (if the program is NOT requesting input yet the key-stroke just goes in a type-ahead buffer and nothing special happens).

If the user has moved the caret to other than at the end of the buffer then the window is never scrolled by cwin, but in cases when it might be interesting to scroll it the title text of the window is updated to give the user a clue to that fact.

8 Print and other operations

The regular PRINT item on the menu should just print the whole of the contents of the text buffer. It will apply a fixed with limit and truncate any material that spills off to the right. It will pack lines onto pages in a simple-minded manner. PRINT-SELECTION behaves the same way but only processes text within the selected region of text. The READ menu operation will insert text of the form

```
IN "<filename>";
```

into the type-ahead buffer.

9 Implementation status

User-controlled scrolling: OK

Selection using the mouse: OK

CUT: OK, although too much may be re-painted after a CUT that spans several lines of text

Program output: OK, but optimisation and find control over when the screen gets re-painted will need further work, I expect.

Keyboard input: Only handled when the caret is at the end of the text buffer. Even in that case DELETE stops deleting at the end of the final line rather than deleteing back further, and deletion of prompts as single items is not implemented. INsertion into the middle of the buffer is not finished.

Paste: Not implemented at all, but expected to be a fairly easy follow-on from keyboard input.

UNDO: Nothing done at all yet, and will need re-work of other buffer update operations to preserve information.

Print: Nothing done yet, and the issue of line width of the paper used relative to the current window size is unclear.

Other operations: READ is not done yet, but is probably easy.