Keyboard & mouse input control sequences – modernised ECMA-48 (ISO/IEC 6429) user input

Kent Karlsson
Stockholm, Sweden
2023-01-13

EARLY DRAFT (even the title may change)

1 Introduction

The latest official edition of ECMA-48, the 5th edition, is from 1991, over 30 years ago. Much has happened in the area of user interface during that time. New widespread conventions have arisen, also in the area of keyboard input and also the use of "mice", touchpads, and most lately touch-screens (including handheld devices) that now are nearly ubiquitous on smartphones and "tablets". Some of this has been built into applications that (at least in part) are based on ECMA-48. For instance, that "modifier keys" (shift key, etc.) may affect how the cursor movement by arrow keys are interpreted. For instance, pressing a Ctrl key while pressing a left arrow key will move the cursor to the left not just character by character (or rather, combining sequence by combining sequence), but word by word (for some definition of what is a "word" in this context). And there are also ways of handing "mouse" input in an otherwise ECMA-48 context, at least partially. But for the even newer input methods (like touchscreens) there is as yet no way at all to handle "pointer input" from those in an ECMA-48 manner. This means that applications that are based on ECMA-48 for its keyboard, mouse, etc. input cannot take advantage of many these newer developments. And to the extent that these developments are supported, some of them are not even done in a way that even can be standardised in the ECMA-48 framework.

ECMA-48 tries to cover several disparate areas, as we see them today. And that, unfortunately, without doing a proper division of the control sequences (and escape sequences) it covers. In *Text styling control sequences – modernised ECMA-48 (ISO/IEC 6429) text styling* we cover text styling and propose extensions to cater for many styling variants now in common use.

Here we propose several modifications and extensions of the ECMA-48 keyboard input control sequences to cater for post-5th edition developments w.r.t. keyboards as well as having control sequences for mouse/touchpad/touchscreen input (those are post-5th edition input devices). And to do it in ways that fall within the ECMA-48 framework and already-existing control sequences. However, unfortunately, some of the extensions cannot be done in a way compatible with certain already existing implementations. In those cases, we propose suitable extensions for ECMA-48.

In addition, modern systems allow for multiple displays with windows that may cross over several displays. Neither window UI nor multiple displays (for one UI "desktop") were envisioned by the ECMA-48 (5th edition) standardisers. These features mostly affect such things as window manipulation (supported by xterm via private use extensions) and mouse/trackpad/touchscreen handling (in part supported by xterm via private use extensions incompatible with ECMA-48).

Note that these are user input control sequences (key presses and “pointer” movements). None of the control sequences listed in this proposal are suitable to store in a document file (as actual control sequences) in normal circumstances. Several of them are likely to cause havoc if blindly output to an application that directly interprets these ECMA-48 control sequences (like a terminal emulator). Several of the control sequences specified in ECMA-48 are used for redrawing the “terminal screen”, the “display component” in ECMA-48 terminology. The “display component” in a modern setting need not be a “terminal”, and one may use completely different techniques to update the display/window. Control sequences that are used to “edit” a terminal emulator window are out of scope for this proposal. This proposal is about keyboard and mouse input to an application, which includes cursor movements in the “data component”, function key input (such as pressing Ctrl-v to paste in text from a copy-paste buffer).

This is opposed to the control sequences for text styling or controls for math expressions. Those can reasonably be stored directly in a file, there is no requirement for (say) conversion to/from another representation. Thus, the control sequences we deal with in this proposal should ideally be filtered out if they occur in a document and outputting the document to an ECMA-48 interpreting application (the typical such application today is a terminal emulator). When using a different kind of “display component”, then all control sequences must be removed, replacing them with other techniques for achieving a certain display effect (such as bold, blue text, or larger font).

Importantly, note that control sequences purely for the redrawing the “screen” (the “display component”) in a terminal emulator is out of scope for this paper. That is even tough we will touch upon them sometimes. But that is just because some control sequences can be used both from the “display component” (typically the keyboard) and to the “display component” (typically a terminal emulator window).

The keyboard and mouse control sequences will likely continue to be used only by terminal emulators and programs that are run via terminal emulators, while other systems continue to use other keyboard/mouse input representations (in particular, keyboard events and mouse events). However, it is technically possible for also for “non-terminal” applications to interpret keyboard input control sequences, though unlikely to happen. So in practice, the proposals in this paper are for terminal emulators and programs that are run via terminal emulators. Several of the proposals here are already implemented extensions in that context (and just re-reporting them here). But some are new, in particular for mouse/touchpad/touchscreen pointing input.

2 C0, C1 characters, escape and control sequences (limited)

Originally, the syntax for escape sequences and control sequences were formulated as byte sequences (or even bit sequences) in ECMA-48. But now we use Unicode, with several character representations that do not conserve the representation as byte sequences. So here we formulate the “overall” syntax (a regular expression), not as *byte* sequences, but as *character* sequences.

The general (almost catch-all) syntax for ECMA-48 escape sequences, control sequences and control strings are as follows, generalized to specify Unicode (or ISO/IEC 10646) characters rather than bytes, but excluding code page shifting controls and escape sequences as well as excluding most device controls, except **DC1** and **DC3**. The latter two, though private use, are de facto standard for xterm and other terminal emulators. **DCS** is start of device control string, *was* used in xterm, when received from the remote side, to “program” the “function keys”, but it is better to use the standard FNK

(FUNCTION KEY) control sequences that we will extend to also cover “ctrl-<key>” which is more popular today:

c0-control-character ::= // here we include four device control characters that are still viable
[\u0001-\u0004\u0007-\u000D\u0011\u0013\u001A-\u001F] // has exclusions
SIGINT ::= \u0003 // end of text (EOT); but for Linux/Unix input from character devices changed to
// mean “generate a SIGINT to the process that currently would read from that character device”
EOF ::= \u0004 // end of transmission; but for Linux/Unix input from character devices means
// to “effectuate an EOF to the process that currently would read from that character device”
BEL ::= \u0007 // only has meaning when sent *to* a terminal emulator, not as keyboard input
XON ::= \u0011 // actually DC1, a private use device control character, but de facto XON when
// received *from* a terminal emulator
XOFF ::= \u0013 // actually DC3, a private use device control character, but de facto XOFF when
// received *from* a terminal emulator
ESC ::= \u001B // limit to (**ESC** [\u0040-\u004C\0050-\0052\u0056-**\u0063**])?:
escape-sequence ::= **ESC** [\u0030-\u004C\0050-\0052\u0056-\u007E] // has exclusions

c1-control-character ::= [\u0080-\u008C\u0090-\0092\u0096-\u009F] |
ESC [\u0040-\u004C\u0050-\0052\u0056-\u005F] // has exclusions
cf-control-character ::= <characters with general category Cf, Zl, Zp>

SCI ::= (**ESC** \u005A|\u009A) // consider Unicode surrogates, non-characters, and Cf excluded:
sci-sequence ::= **SCI** [\u0001-\u0003\u0008-\u000D\u001C-\u007E\u00A0-\U10FFFD]

CSI ::= (**ESC** \u005B|\u009B) // using [\003C-\u003F] *first* or term. [\u0070-\u007E] is private use:
control-sequence ::= **CSI** [\u0030-\u003F]* [\u0020-\u002F]* [\u0040-\u007E]

Somewhat annoyingly, ctrl-c on a terminal emulator generates \u0003, causing a SIGINT, while for other programs today, it usually is a character function key used for “copy selection to copy-paste buffer”. To overcome this a bit, Meta-c (Win-c, command-c) can still be used for the latter, also for terminal emulators.

Since this is a proposed update, there will be no private use control sequences in the proposal. We will use some of \003C-\u003F *non-first* after the **CSI**, for negation (**=**), and decimal marker (**?**). We cannot use HYPHEN-MINUS (-) nor FULL STOP (.) due to the general syntax for control sequences; they may occur just before the final character, but that is unsuitable for this.

Note that we refer to characters, and their Unicode scalar values, not bytes. This means that the control sequences can also be used also with EBCDIC code pages (may need custom mapping for some “national variants”, but that mapping is out of scope for this proposal), as well as Unicode encodings (UTF-8 or UTF-16).

3 Goals and non-goals for this update proposal

Goals for the additions and clarifications in this proposal:

- Cover control sequences for (ordinary computer) keyboard input from various keys, or key combinations, that do *not* generate “printable” characters (or character sequences). Like arrow keys, function keys, … (Following ECMA-48, keys that generate characters simply generates those characters; there is no “event” structure in ECMA-48.)
- Be compatible with ECMA-48, while allowing common or reasonable extensions, esp. w.r.t. arrow keys and other keys “normally” found on present-day computer keyboards.

- Add extensions for mouse, touchpad and touchscreen input.
- Make reasonable adjustments adapting to common implementations.

Non-goals for the additions and clarifications:

- Make extensions that are not compatible with ECMA-48 basics.
- Handle input from non-keyboard buttons (or unusual buttons on the keyboard).
- Handle input from joysticks, 3D-"pens", sensors (position, temperature, RFID locks, pressure, gyroscopes, …), scanners, cameras, microphones, eye trackers, …
- Handle "virtual reality" or "augmented reality" input; like from virtual controls (buttons, levers, joysticks, "doors", "items" (that can be picked up), …) displayed to the user to interact with.
- Handle forms input.
- Handle file reading input.

4 "Data component" and "display component"

ECMA-48 uses an unfamiliar terminology in using the terms "data component" and "display component". It is not as strange as it might sound, and it is a perfectly usable terminology also in today's setting.

- The "data component" is the actual "open document". In ECMA-48 it usually refers to some kind of text-based document (referring to what is sometimes called the "backing store" for the text document), but is more general. It includes such things as "command line history". It includes the edit position, which may be moved around in the open document, and is where characters may be inserted or deleted (if not a read-only "document"). In more modern contexts (than the 5th edition…), this covers also text substring selection. It is even more general than that, in modern setting, and includes any kind of document, such as a video editor, where the "backing store" is the "video" (including sound, as well as any text overlays and subtitles), just to give an example.
- The "display component" is where the "open document" is displayed. It may be a terminal emulator "screen" (with cursor position). But it should also be seen as covering the case where the "open document" is displayed in other ways, typically an edit window in a "windows" based environment, or any other kind of application displaying in a (or several) window(s). Some of these "display components", but far from all, may be suitable to also get "display commands" in the form of a stream of text that includes control sequences. These latter applications are typically terminal emulators.

There is no requirement that both the "data component" and "display component" accept ECMA-48 (like) data. However, that is in practice always the case for terminal emulators, and terminal emulators only. Note though that ECMA-48 control sequences may be used to style text in a (formatted) text file (see *Text styling control sequences – modernised ECMA-48 (ISO/IEC 6429) text styling*). However, the control sequences that are the subject of *this* proposed update, are for sending UI commands from the "display component" (in particular the associated keyboard and mouse) to the "data component".

In this proposed update, we consider only the communication *from* the "display component" *to* the "data component" (not the other way around). We will cover keyboard input (apart from generated printable characters, as they stand for themselves, via a keyboard layout mapping), as well as mouse/ touchscreen (including handheld units) input. Even some UI window (or part of window) changes from the “display component” side. But not such things as eye tracking, virtual reality, or voice user input.

All the control sequences covered by *Text styling control sequences – modernised ECMA-48 (ISO/IEC 6429) text styling* (with the exception of UI styling) are what we today would regard as data format for styling and layout of text. That subset is similar to such things as HTML, RTF, docx (MS Word XML-based format), “markdown” and others. Each with different capabilities and properties, of course, they are in no way equivalent, but are similar in nature.

This paper considers control sequences (and some escape sequences) for what we now would regard as keyboard and mouse events. But for the purpose of ECMA-48, such an “event” is coded as a “printable” characters (representing themselves), “control” characters, plus certain escape sequences and control sequences (the topic of this paper). It is a bit unfortunate that there is no easy and direct distinction (other than control sequence enumerations) between “suitable for document formatting” and “input from keyboard/mouse”. But that is how ECMA-48 is designed.

For terminal emulators (which is the current major type of implementation that implements several features of ECMA-48) these sequences of characters are sent “in” via a “character device”. In Unix/Linux: /dev/tty. But the character sequences are usually read as standard input. This is often targeted at shell programs (like *bash*), text editor programs (like *vi*), or text display programs (like *less*), but can go to any other program that reads input from a terminal.

Most “keyboard events” will then simply be ordinary “printable” characters (without “wrapping” them into “events”), but some will be generating control sequences (or escape sequences), like for the arrow keys. Here (like done for *xterm*) we extend this to cover mouse “events” as control sequences (though in a different, more ECMA-48, way than done in *xterm*). The mouse “event” control sequences we will propose here do not follow the *xterm* way of doing this, since the (current) *xterm* way is not only non-ECMA-48 in nature, but also too limited given modifier keys, touchpads and touchscreens.

Note that the control sequences dealt with in this paper should not be stored in documents (even if they otherwise use ECMA-48 based styling), and if they occur anyway, they should be filtered away. Having them in string literals in programs, using the escape character mechanism of the programming language (like \e), is of course ok, programs need to be able to match incoming control sequences to stored patterns.

For “events”, à la ECMA-48, they can be used as replacement for the event system now often used for keyboard and mouse/touchpad/touchscreen input to a GUI-based program.

And then we have what is output to the user (i.e., output to the “display component” to be seen by the user). There is *no* requirement in ECMA-48 that that be in the form of characters and control sequences (or escape sequences). But that is how *terminal emulators* work. The presentation to the user may instead well be in the form of a graphical user interface. Especially for formatted text file format using ECMA-48 text styling, the presentation may be as a “normal” GUI-based text editor or text display program.

4.1 “Active data position” and “active presentation position”

Corresponding to the “data component” and “presentation component” ECMA-48 talks about two active positions.

- The “active data position” is the edit position (in a (text) document that can be edited). Post-5th edition of ECMA-48, this has been generalised to enable having a substring as the target of edit operations. This can also be used for documents that cannot be edited, but allows for copying a substring to the copy-paste buffer; another post-ECMA-48 5th edition invention.

- The “active presentation position” is much more orientated towards how terminals worked and terminal emulators work. In a (non-terminal emulator) GUI program, one would use graphical operations to update the display. In a terminal emulator, however, the emulator would get a stream characters and control sequences to do the display updates (they would in turn be converted to graphical operations to actually do the display updates in GUI fashion, but that is internal to the terminal emulator). The control sequences used for these terminal screen updates do not seem to need any update compared to ECMA-48 5th edition.

We will focus on the “data component” (and “active data position”) and regard the traditional handling of the “presentation component” as good enough as it is, and thus out of scope for this proposed update. But there are some things also for the “display component” side that we will have in scope, mostly because they are post-5th edition. Those are of course only aimed at terminal emulators, not other kinds of systems that can use ECMA-48 control sequences.

4.2 Modifier key information

Regarding keyboard (and mouse/touchpad) input, modern UIs act not only on which key is pressed for arrow keys, function keys and mouse/touchpad key click input. Modern UIs also modify the action implied by these keys through the use of “modifier keys”. The modifier keys are: the shift keys (both equivalent), the Alt(Gr)/Option keys, the Ctrl key(s), the “meta” key(s) (the “meta” keys are called command keys on Mac, and Windows key(s) on Windows/Linux). An unfortunate development is that for Windows and Linux the Ctrl key(s) are used for the same purpose as the command (“meta”) keys on Mac. To alleviate that problem a bit in ECMA-48 contexts, we strongly recommend treating Ctrl and “meta” keys as equivalent, even though we for compatibility with existing practice allow for distinctive coding of them in the modifier keys code proposed below. We also very strongly recommend treating both option/alt(gr) keys as equivalent, and we make no distinction between them in the proposals below (that goes for both the modifier keys code, and their function as level 3 and 4 (with shift) select for graphical characters that can be generated from the keyboard).

We will use the modifier keys codes as listed below. These codes are taken from the existing convention in use in *xterm*. We will use the variable *m* for modifier key values. Modifier keys can be combined, and it is recommendable to then “combine” the semantics of the modifications. To represent the combination in the code, we use bit *or*, and then add 1 to that (that is how the *xterm* modifier key codes are constructed). Note that if a particular modifier key bit has no given interpretation, the rest of the bits are still interpreted. E.g., if “Alt” has not been given a “meaning”, then “Shift + Alt” is interpreted the same as “Shift” for modifier parameter purposes (this is not required for their “normal” meaning when generating letters/digits/punctuation/symbols). If two (or more) modifier keys are given a semantics, the combination of those keys should have a semantics that is the combination of the semantics of the individual modifier keys. “Meta” and “Ctrl” should be equivalent for ECMA-48; indeed, when generating “character function key” control sequences, “Meta” and “Ctrl” should be equivalent. Also when interpreting “Meta” and “Ctrl” modifier key values for cursor movement and mouse controls sequences, they should be interpreted the same. To actually generate some C0 control characters (used by Unix/Linux terminal/tty drivers, for such things as “simulate EOF”, “send SIGINT”, and more), double-click (and hold down) Ctrl key and a character key related to the C0 control code desired.

- **0** No modifier data available (from touchscreen; or default for from data component)
- **1** No modifiers (default for presentation component, i.e, keyboard/mouse/touchpad)
  arrow keys/mouse: move by character (or line)
- **2** Shift (left or right or both)
  arrow keys/mouse: extend/shrink selection by character (or line)
- **3** Alt (left or right or both, includes “AltGr” which we here consider equivalent)
- **4** Shift + Alt (any combination of left and right)
- **5** Control (left or right or both) arrow keys/mouse: move by word boundary
- **6** Shift + Control (any combination of left and right)
  arrow keys/mouse: extend/shrink selection by word
- **7** Alt + Control (any combination of left and right)
- **8** Shift + Alt + Control (any combination of left and right)
- **9** “Meta” (a.k.a “Command”, a.k.a. “Windows”; left or right or both)
- **10** “Meta”+Shift (left or right or both)
- **11** “Meta”+Alt (any combination of left and right)
- **12** “Meta”+Shift + Alt (any combination of left and right)
- **13** “Meta”+Control (any combination of left and right)
- **14** “Meta”+Shift + Control (any combination of left and right)
- **15** “Meta”+Alt + Control (any combination of left and right)
- **16** “Meta”+Shift + Alt + Control (any combination of left and right)

For the purpose of modifying arrow keys, non-character function (F1, F2, …) keys, and mouse/touchpad events (but not necessarily for other uses):

- On Windows keyboards, Alt and AltGr are the same (use either one or both simultaneously)
- “Meta” on Windows keyboards is the Windows keys
- “Meta” on Mac keyboards is the Command keys

The cursor movement control sequences, scrolling control sequences and many of the mouse/touchpad control sequences (that we will introduce below) will now have an argument that indicates which “modifier” keys were also pressed.

- For these control sequences, used with a text document, the Shift modifier changes the semantics to instead of just moving the cursor, the text selection is extended or shrunk (moving just the “secondary cursor position”).
- For these control sequences, used with a text document, the Ctrl modifier changes the semantics of the character movements to move “word-by-word” (for an implementation defined meaning of “word”). For the line movements, there is no modified semantics.
- The “Alt” modifier for these control sequences is implementation defined.
- If an “unmodified” cursor movement is done when there is a (non-empty) text selection, the cursor is moved relative to the “second cursor position” of the selection, while releasing the selection.
- A sequence of cursor movement control sequences that move between lines (but may be interspersed with scrolling control sequences), but none of which also move to the beginning position of the target line, uses two character positions: 1) the original (of the sequence) character position, and 2) the closest available character position (to the currently original character position, the latter is the target character position) in the target line.

- ❖ If the edited text is not actually a text document, the line cursor movements may be suitably reinterpreted. E.g., in an interactive command interpreter, line cursor movements may move between commands saved in a history list, and when edited or executed, the "current" command line is copied as the new command line. (This is commonly the case in interactive command-line interpreters.)
- ❖ For cursor moves that move between lines, the character position relative to the margins is not moved. Note that this refers to "physical" position (distance in fractions of em, picking the closest available possible cursor position), *not* number of characters or number of combining sequences (fonts may be "variable width" and also that SP, HT and HTJ are "variable width", plus that parts of the text may be "stretched" by styling control sequences).

4.3 Movement of text editing point and change of text selection extent

For this section it is extra important to recall that the proposed updates in this paper are limited to keyboard input (and mouse/touchpad/touchscreen input, and some window manipulation), with very few related exceptions. We here do *not* cover *output* to "screens" (here called "tiles"; in a terminal emulator). Hence, any use of (these or other) "movement" control sequences for outputting any kind of text display to a terminal emulator tile are *out of scope* for this paper. The descriptions are still very much oriented towards text editing, rather than other applications. Other applications may interpret these control sequences (as keyboard input) in completely different ways, but still should have some relation to notions of up/down, left/right. The scrolling control sequences (as keyboard or mouse input) may also be suitably reinterpreted for the application at hand if it is not a text-oriented application.

Even so, the interpretation here not only has some (some already implemented in xterm and its "clones") extensions, but most probably also give a different interpretation than originally intended. But we need to adapt the interpretation both to current actual practice of using these control sequences (at least in part), and to how insertion point movement is implemented in systems not based on ECMA-48 and also to different ways of handling such movement alternatives in the presence of bidi reordering as well as the more common automatic line breaking, and different line and character progression directions.

The insertion point movements (not counting scrolling) are divided into two sets, the ones with "CURSOR" in the name, and the ones with "CHARACTER" in their names. Perhaps, not clear, the intent was that the "CHARACTER" ones were supposed to change the insertion point on the "remote" side (of a terminal (emulator)), "data component" and the "CURSOR" ones at the local side (of a terminal (emulator)), "presentation component". But that is not how it has materialised in actual implementations (of terminal emulators). In practice it is only some of the "CURSOR" ones that are (explicitly) implemented, and they are sent both to the remote side and to the local side (of a terminal emulator). Hence, we will ignore the distinction between "data component" and "display component", as that distinction has not been realised, at least not in the way originally intended.

New here, is that we allow first argument to the movement and scrolling control sequences to be zero or negative (**=** as postfix negation sign). For keyboard input, however, they will be **1** or **=1** (minus one). The scrolling control sequences may have fractional arguments (**?** as decimal marker) and may also have unit codes (**:***u*). (Compare similar extensions given in *Text styling control sequences – modernised ECMA-48 (ISO/IEC 6429) text styling*.) For all of these insertion point movements, the default for *n* is **1** and the default for *m* is **1**.

The lines (of characters) here are lines *after* automatic line breaking. Automatic line breaking effectively inserts LS characters (before bidi, if bidi is enabled) based on the (final, after kerning and ligature formation) display width of the characters. This must be done before bidi processing (if enabled), so that bidi does not ruin the display of the text completely. Position movement, if bidi is enabled, can be done either before bidi, in which case a contiguous selection (in the backing store) may be split into several displayed selections, or after bidi, in which case a contiguous displayed selection actually may be several substrings in the backing store. An alternative is to enlarge the selection so that it is contiguous in both aspects (see https://patents.google.com/patent/US9696818B2/en). Which method is used (if bidi is enabled) is implementation defined. A similar, but smaller scale, issue happens with some Indic scripts that have vowel marks that reorder before the base character. For the Indic scripts, the latter method (extend selection) is probably the best.

By "character position" is meant a point in the text that can be an insertion position. For instance, mostly combining sequences do not have any insertion position inside them. Exactly which positions are insertion positions is implementation defined.

The "LINE/CHARACTER POSITION" control sequences apparently were intended to be sent from the "display component" to the "data component" in order to move the insertion point (or by extension the text selection). But it did not turn out that way. Instead, the arrow keys on the keyboard (of the "presentation component") send "CURSOR" control sequences (which in origin apparently were only to be sent *from* the "data component" *to* the "presentation component"; a use that is out of scope for this proposed update). So in practice the following four control sequences are not used directly, but we will give (conceptual) mappings to them from four "CURSOR" control sequences.

- **CSI** [*n*][**;***m*]**j** HPB - CHARACTER POSITION BACKWARD. **CSI** *n*=[**;***m*]**j** is the same as **CSI** *n*[**;***m*]**a**.
  HPB causes the second or both active data positions to be moved by *n* character or word (depending on modifier value, *m*) positions in the data component in the direction opposite to that of the character progression.

- **CSI** [*n*][**;***m*]**a** HPR - CHARACTER POSITION FORWARD. **CSI** *n*=[**;***m*]**a** is the same as **CSI** *n*[**;***m*]**j**.
  HPR causes the active data position to be moved by *n* character or word (depending on modifier value, *m*) positions in the data component in the direction of the character progression.

- **CSI** [*n*][**;***m*]**k** VPB - LINE POSITION BACKWARD. **CSI** *n*=[**;***m*]**k** is the same as **CSI** *n*[**;***m*]**e**.
  VPB causes the active data position to be moved by *n* line positions in the data component in a direction opposite to that of the line progression. No-op if *n* is 0 or position reaches the first line in the document. Effectively, repeating CHARACTER POSITION BACKWARD (with default first argument, and *m* as second argument), until coming as close as possible to the same position (pixel-wise) in the target line as the target position. The character progression target position (pixel-wise) does not change with LINE POSITION FORWARD and LINE POSITION BACKWARD, but does with every other operation except scrolling operations.

- **CSI** [*n*][**;***m*]**e** VPR - LINE POSITION FORWARD. **CSI** *n*=[**;***m*]**e** is the same as **CSI** *n*[**;***m*]**k**.
  VPR causes the active data position to be moved by n line positions in the data component in a direction parallel to the line progression. No-op is *n* is 0 or position is at the last lin in the document. Effectively, repeating CHARACTER POSITION FORWARD (with default first argument, and *m* as second argument), until coming as close as possible to the same position

(pixel-wise) in the target line as the target position. The character progression target position (pixel-wise) does not change with LINE POSITION FORWARD and LINE POSITION BACKWARD, but does with every other operation except scrolling operations.

The “CURSOR … TABULATION” control sequences were apparently intended to (only) be sent *from* the “data component” *to* the “display component” (a use that is out of scope for this proposed update proposal), the original specification saying “presentation position”. But in practice, some of them are also sent *from* the keyboard of the “display component” *to* the “data component” (and that use is in scope for this proposed update). They can be used to move the insertion point between tab positions, but in tables move between table cells. Note that these are different from inserting any kind of TAB character; they move the insertion point (or expand/contract the text selection, an idea that did not exist when ECMA-48 5th edition was written).

- **CSI** [*n*][**;***m*]**Z** CBT - CURSOR BACKWARD TABULATION. **CSI** *n*=[**;***m*]**Z** is the same as **CSI** *n*[**;***m*]**I**.
  CBT causes the active *data* position to be moved to the closest available, pixel-wise, character position corresponding to the *n*-th preceding character tabulation stop in the presentation component, according to the character progression direction. Stops at beginning of line as displayed (just before first character displayed in the line) in the character progression direction, even if there is no tab stop there. This control sequence moves to the closest available position to targeted tab stop.

- **CSI** [*n*][**;***m*]**I** CHT - CURSOR FORWARD TABULATION. **CSI** *n*=[**;***m*]**I** is the same as **CSI** *n*[**;***m*]**Z**.
  (**I** is capital i) CHT causes the active *data* position to be moved to the closest available, pixel-wise, character position corresponding to the *n*-th following character tabulation stop in the presentation component, according to the character progression direction. Stops at end of line as displayed (just after last character in the displayed line, not counting NLF characters) in character progression direction, even if there is no tab stop there.

- **CSI** [*n*][**;***m*]**Y** CVT - CURSOR LINE TABULATION.
  CVT causes the active *data* position to be moved to the closest available, pixel-wise, character position of the line corresponding to the *n*-th following line tabulation stop in the presentation component.
  <u>If line tabulation is *not* implemented&enabled</u>: **CSI** [*n*][**;***m*]**Y** is the same as **CSI** [*n*][**;***m*]**e**.
  <u>If line tabulation is implemented&enabled</u>: **CSI** [*n*][**;***m*]**Y** is implementation defined.

Like the “CURSOR … TABULATION” control sequences, the “CURSOR UP/DOWN/LEFT/RIGHT” were in origin intended to (only) be sent *from* the “data component” *to* the “display component” (a use that is out of scope for this proposed update). But in practice, they are also sent *from* the keyboard of the “display component” *to* the “data component” (and that use is in scope for this proposed update). We will (conceptually, no requirement to actually do the mapping) map them to the four “LINE/CHARACTER POSITION” control sequences.

- **CSI** [*n*][**;***m*]**A** CUU - CURSOR UP. **CSI** *n*=[**;***m*]**A** is the same as **CSI** *n*[**;***m*]**B**.
  If the line progression direction is top to bottom, then this control sequence is the same as LINE POSITION BACKWARD. If the character progression direction is top to bottom, it is the same as CHARACTER POSITION BACKWARD.

- **CSI** [*n*][**;***m*]**B** CUD - CURSOR DOWN. **CSI** *n***=**[**;***m*]**B** is the same as **CSI** *n*[**;***m*]**A**.
  If the line progression direction is top to bottom, then this control sequence is the same as LINE POSITION FORWARD. If the character progression direction is top to bottom, it is the same as CHARACTER POSITION FORWARD.

- **CSI** [*n*][**;***m*]**D** CUB - CURSOR LEFT. **CSI** *n***=**[**;***m*]**D** is the same as **CSI** *n*[**;***m*]**C**.
  If the line progression direction is right to left, then this control sequence is equivalent to LINE POSITION FORWARD. If the line progression direction is left to right, then this control sequence is equivalent to LINE POSITION BACKWARD. If the character progression direction is left to right, then this control sequence is equivalent to CHARACTER POSITION BACKWARD. If the character progression direction is right to left, then this control sequence is equivalent to CHARACTER POSITION FORWARD.

- **CSI** [*n*][;*m*]**C** CUF - CURSOR RIGHT. **CSI** *n***=**[;*m*]**C** is the same as **CSI** *n*[;*m*]**D**.
  If the line progression direction is right to left, then this control sequence is equivalent to LINE POSITION BACKWARD. If the line progression direction is left to right, then this control sequence is equivalent to LINE POSITION FORWARD. If the character progression direction is left to right, then this control sequence is equivalent to CHARACTER POSITION FORWARD. If the character progression direction is right to left, then this control sequence is equivalent to CHARACTER POSITION BACKWARD.

CHARACTER POSITION FORWARD/BACKWARD should be counted relative to the stored text, *before* bidi processing (on the data component side). LINE POSITION FORWARD/BACKWARD should be counted *after* automatic line breaking (nominally inserting LS, so that bidi processing works as expected; on the data component side). Handling of automatic line breaking or bidi on the display component side is implementation defined and may cause unexpected behaviour.

4.4 Scrolling

The first argument to the scrolling control sequences may be zero or negative (**=** as postfix negation sign). The scrolling control sequences may have fractional arguments (**?** as decimal marker) and may also have unit codes (**:***u*), and the default for the unit here is "em of the default font" (code **0**).

- **CSI** [*n*[**:***u*][**;***m*]]**S** SU – SCROLL UP
  Default for *n* is **1**, default for *u* is 0 (em of the *default* font size), default for *m* is **1**. SU causes the data in the presentation component to be moved by n (can be fractional or negative) ~~line positions~~[steps of unit *u*, default unit code 0, em of default character size] if the line orientation is horizontal, or by *n* (can be fractional or negative) ~~character positions~~[steps of unit *u*, default unit code 0, en of the default character size] if the line orientation is vertical, such that the data appear to move up; where *n* equals the value of Pn.
  The active presentation position is not affected by this control function.

- **CSI** [*n*[**:***u*][**;***m*]]**T** SD – SCROLL DOWN
  Default for *n* is **1**, default for *u* is 0 (em of the *default* font size), default for *m* is **1**. SD causes the data in the presentation component to be moved by n (can be fractional or negative) ~~line positions~~[steps of unit u, default unit code 0, em] if the line orientation is horizontal, or by n (can be fractional or negative) ~~character positions~~[steps of unit u, default unit code 0, en]if

the line orientation is vertical, such that the data appear to move down; where n equals the value of Pn.
The active presentation position is not affected by this control function.

- **CSI** [*n*[**:***u*][**;***m*]] **SP A** SR – SCROLL RIGHT
  Default for *n* is **1**, default for *u* is 0 (em of the *default* font size), default for *m* is **1**. SR causes the data in the presentation component to be moved by n character positions if the line orientation is horizontal, or by n line positions if the line orientation is vertical, such that the data appear to move to the right; where n equals the value of Pn.
  The active presentation position is not affected by this control function.

- **CSI** [*n*[**:***u*][**;***m*]] **SP @** SL – SCROLL LEFT
  Default for *n* is **1**, default for *u* is 0 (em of the *default* font size), default for *m* is **1**. SL causes the data in the presentation component to be moved by n character positions if the line orientation is horizontal, or by n line positions if the line orientation is vertical, such that the data appear to move to the left; where n equals the value of Pn.
  The active presentation position is not affected by this control function.

- **CSI** [*n*[**;***m*]] **SP R** PPB – PAGE POSITION BACKWARD
  Default for *n* is **1**, default for *m* is **1** (no modifiers). PPB causes the active data position to be moved in the data component to the corresponding character position on the n-th preceding page, where n equals the value of Pn.
  Note that **CSI** *n*=[**;***m*] **SP R** is equivalent to **CSI** *n*[**;***m*] **SP Q**.

- **CSI** [*n*[**;***m*]]**V** PP – PRECEDING PAGE
  Default for *n* is **1**, default for *m* is **1** (no modifiers). Scroll *n* "window-fulls" opposite to the line progression direction. For extended PP, *n* may be fractional or negative.
  Note that **CSI** *n*=[**;***m*]**U** is equivalent to **CSI** *n*[**;***m*]**V**.

- **CSI** [*n*[**;***m*]] **SP Q** PPR – PAGE POSITION FORWARD
  Default for *n* is **1**, default for *m* is **1** (no modifiers). PPR causes the active data position to be moved in the data component to the corresponding character position on the n-th following page, where n equals the value of Pn.
  Note that **CSI** *n*=[**;***m*] **SP Q** is equivalent to **CSI** *n*[**;***m*] **SP R**.

- **CSI** [*n*[**;***m*]]**U** NP – NEXT PAGE
  Default for *n* is **1**, default for *m* is **1** (no modifiers). Scroll *n* "window-fulls" in the line progression direction. For extended NP, *n* may be fractional or negative.
  Note that **CSI** *n*=[**;***m*]**V** is equivalent to **CSI** *n*[**;***m*]**U**.

- **CSI** [*n*[**;***m*]] **SP P** PPA – PAGE POSITION ABSOLUTE
  Default for *n* is **1** (referring to the first page), default for *m* is **1** (no modifiers). PPA causes the active data position to be moved in the data component to ("as close as possible") the corresponding character position on the *n*-th page, where n equals the value of Pn. A 0 or

negative *n* counts from the last page. Thus **CSI 0 SP P** refers to the last page (or “window-full” if there are no pagination) of the data component.

4.5 Interrupt and erase

Some common keys have functionality that that is covered by ECMA-48, 5th edition.

- **ESC a** INT – INTERRUPT Escape sequence. Should be generated by the ESC *key*, that key should not generate **ESC** alone. This key is commonly used for a “cancel”/“interrupt” functionality (for the data component), so the ESC *key* is actually a dedicated function key for INT. *(By default, to send an actual C0 control code using the ctrl key, double-click (and hold down) the ctrl key plus the corresponding letter; this way most users get the functionality expected from e.g. ctrl-c, while still allowing easy access to the Unix/Linux functionality of interrupting a program via the keyboard (ctrl-ctrl-c). Most users find ctrl-c interrupting a program highly annoying when expecting to paste in something.)*

- **CSI =***n*[**;***m*]**X** ECH – ERASE CHARACTER with negative first argument; “Erasing backspace”. (Requests erasing backwards in the character advance direction in the data component. But if there is a (text) selection, erase the selection); may erase other than *n* “characters” with modifiers (erasing backward up to *n*-th preceding word boundary if *m*=**5**). (“Backspace” key should generate this with *n*=1.)

- **CSI** [*n*][**;***m*]**X** ECH with non-negative first argument (default **1**); “Erasing delete” (requests erasing forwards in the character advance direction in the data component. But if there is a (text) selection, erase the selection); may erase other than *n* “characters” with modifiers (erasing backward up to *n*-th following word boundary if *m*=**5**). (“Delete” key should generate this with *n*=1.)

~~4.6 Status requests and responses~~

~~These control sequences (request and response) are basically intended only for terminal emulators. They are hardly useful for any other type of implementations.~~

- ~~DSR – [REQUEST] DEVICE STATUS REPORT~~
  ~~Notation: (Ps)~~
  ~~Representation: CSI Ps 06/14~~ ~~**CSI 6n**~~ ~~**CSI 6!n**~~
  ~~Parameter default value: Ps = 0~~
  ~~DSR is used either to report the status of the sending device or to request a status report from the receiving device, depending on the parameter values:~~
  ~~6 a report of the active presentation position or of the active data position in the form of ACTIVE POSITION REPORT (CPR) is requested. [other param values: DEPRECATE]~~
  ~~*--> ask for screen resolution (pixels per cm) and screen or window size in cm.*~~

- ~~CPR – [GIVE] ACTIVE POSITION REPORT (/P, /D)~~
  ~~Notation: (Pn1;Pn2)~~
  ~~Representation: CSI Pn1;Pn2 05/02~~ ~~**CSI** ***x*****;*****y*****R**~~
  ~~Parameter default values: Pn1 = 1; Pn2 = 1~~

~~If the DEVICE COMPONENT SELECT MODE (DCSM) is set to PRESENTATION, CPR/P is used to report the active *presentation* position of the sending device as residing in the presentation component at the n-th line position according to the line progression and at the m-th character position according to the character path, where n equals the value of Pn1 and m equals the value of Pn2.~~
~~[**CSI** *x*;*y***!R**]~~
~~If the DEVICE COMPONENT SELECT MODE (DCSM) is set to DATA, CPR/D is used to report the active *data* position of the sending device as residing in the data component at the n-th line position according to the line progression and at the m-th character position according to the character progression, where n equals the value of Pn1 and m equals the value of Pn2.~~
~~CPR may be solicited by a DEVICE STATUS REPORT (DSR) or be sent unsolicited. [Note that this is in terms of lines/characters in the display window, after line breaking, and a partial leading line reported as line 0, and a partial leading character is reported as character 0. Also positions outside of the visible window can be reported.]~~

- ~~report screen resolution (px/em), line spacing (in em), tile size (px,px), (no reporting of window size or position, nor of tab size or position) [need to handle multi-screen], fullscreen status?~~
- ~~report rotation (degrees, reported by **CSI 1;***angle***!@**)~~
- ~~report caps-lock/num-loc/other-locks status~~

5 Alphanumeric keyboard non-alphanumeric keys

~~handle locally vs. handle remotely (co-opt EMI/DMI?   ESC ', ESC b)........ By default, those with negative function key parameters, except caps/num lock engage/disengage are send to the remote side, while the others are handled locally.~~

A local system (with the keyboard, if any) is likely to have a number of dedicated function keys that are handled locally. E.g. change screen brightness, change volume, multi-screen mode, and others. Some may be appropriate to send to the remote side, but that is an extension that is implementation defined.

Modifier keys codes (*m* below) here are the same as for cursor movement control sequences.

5.1 Keyboard keys vs. characters/escape sequences/control sequences

- **Tab key**. Sends an **HT** normally (without modifier keys). We here suggest that *Shift-Tab* sends **HTJ** (possibly as **ESC I**), *Ctrl-Tab* to sends **CHT** (), *Ctrl-Alt-Tab* to send **CBT** (),as hinted by the keytop symbol on most keyboards. Note that the two latter are “cursor” movement operations (see above) moving the current position in the “data component”. There should be no insertion of any character for **CHT** or **CBT**.

- **Delete key**. While **DEL** once did have a use ("editing" punched paper cards and punched paper tape), that use is very long obsolete. So, **DEL** is now often given the editing semantics of ERASE CHARACTER (with a **1** as (default) argument). (The semantics of the DELETE CHARACTER control sequence is very dependent on "modes", and we deprecate all modes.) However, the use of the delete key in contexts outside of ECMA-48 depends on modifier keys: *Ctrl-delete* erases forward to a word boundary. **DEL** cannot be extended with modifier information. But ERASE CHARACTER can: **CSI** *n*[**;***m*]**X**, where the *m* gives modifier key info. So to enable extending ECMA-48 to the modern functionality of the delete key, we propose that it actually sends **CSI** *n*[**;***m*]**X** with suitable parameters. (To get that working properly (using terminal emulators) needs programs that interpret such control sequences properly, of course. So, fixing this will not go overnight…)

- **Backspace key**. While **BS** once did have a use (overtyping to create bold on typewriterlike terminals, or overstriking a letter with an accent), that use is long obsolete. So **BS** is now often given the editing semantics of ERASE CHARACTER (with a negative one as argument). (The semantics of DELETE CHARACTER is very dependent on "modes", and we deprecate all modes.) However, the use of the backspace key in contexts outside of ECMA-48 depends on modifier keys: *Ctrl-backspace* erases to a preceding word boundary. **BS** cannot be extended with modifier information. But ERASE CHARACTER can: **CSI** *n***=**[**;***m*]**X**, where the *m* gives modifier key info. So to enable extending ECMA-48 to the modern functionality of the backspace key, we propose that it actually sends **CSI** *n***=**[**;***m*]**X** with suitable parameters. (To get that working properly (using terminal emulators) needs programs that interpret such control sequences properly, of course. So, fixing this will not go overnight…)

- **Escape key**. The modern use of the escape key is *not* to send ESC… Instead, it is to cancel a UI operation (usually) in progress, like closing a popup menu, a popup form window, or exit from full-screen mode, or similar. In ECMA-48 that corresponds the INT control sequence: **ESC a**. (Note that there is no possibility to insert modifier key info.) ~~((*Alt-ESC* can still be a local interrupt… or send something else…))~~

- **Home key**. xterm has this key as **CSI H**, i.e. (expanding the defaults) **CSI 1;1H**. This moves the cursor to the upper left corner of the terminal "screen" (i.e. a command to be sent from the "data component" to the "presentation component"). But that is not at all how "home" is interpreted in modern window-based UIs; where "home" scrolls to the beginning of the document, and it is a command sent from the "presentation component" (and the keyboard part of that) to the "data component". But there is a suitable ECMA-48 control sequence **CSI 1;***m* **SP P** (**SP** is a space). When *m*=**5** (Ctrl, or another modifier code that includes Ctrl), calling for a mild confirmation (Ctrl), since this may scroll very far; if *m*=**1** this should be a no-op). Note that we here have extended the control sequence to allow for modifier keys. ((Alt-Home can be a local to-beginning of scroll buffer….))

- **Page up key**. This corresponds to **CSI 1;***m***V**, PP – PREVIOUS PAGE, where "page" is interpreted as "screen-full" (when *m*=**1**), even for paginated documents. But when *m*=**5** (Ctrl, or another code that includes Ctrl), then go by true page for paginated documents. ((Alt-PageUp can be a local previous screenfull of scroll buffer….))

- **Page down key**. This corresponds to **CSI 1;***m***U**, NP – NEXT PAGE, where "page" is interpreted as "screenfull" (when *m*=**1**), even for paginated documents. But when *m*=**5** (Ctrl, or another code that includes Ctrl), then go by true page for paginated documents. ((Alt-PageDown can be a local next screenfull of scroll buffer….))

- **End key**. In modern window-based UIs "end" scrolls to the end of the document. This corresponds to the here extended ECMA-48 control sequence **CSI 0;***m* **SP P**. When *m*=**5** (Ctrl, or another modifier code that includes Ctrl), calling for a mild confirmation (Ctrl), since this may scroll very far; if *m*=**1** this should be a no-op. Here we also made the extension to let **0** and negative first arguments count pages "up" from the end of the document. ((Alt-End can be a local to end of scroll buffer….))

- **Return key**. Nominally, this sends a CR character. But most systems at some point (before or after turning the keystroke to a CR) map it to an LF or to a CRLF. In a few systems it may still send CR, and other few systems it may send NEL. That this has not been "fixated" is an annoying problem that is still with us. Further, *Shift-Return* should send a VT (usually interpreted as LS or a CRVT), and *Ctrl-Return* may send an FF (or a CRFF). We here suggest that *Alt-Return* send a PS, and *Alt-Shift-Return* send an LS.

- **Shift keys**. Used as a modifier key for arrow keys and some other keys. Used to access "level 2" and "level 4" (together with an Alt key) for printable character keys.

- **Alt keys**. Used as a modifier key for arrow keys and some other keys (for both Alt keys). Used to access "level 3" and "level 4" (together with a Shift key) for printable character keys (usually only works for the left Alt key (AltGr) for Windows). Note that Alt-<char key>, without Ctrl/"Meta", never generates a FNK control sequence.

- **Ctrl keys**. Used as a modifier key for arrow keys and some other keys. Used to make "character-function-keys" for character keys. When double-clicked (and held down), and only then, used to generate some C0 control characters (for TTY-based input that abuses some C0 control codes for special purposes). Character function key control sequences (**CSI** *n*= **SP W**) cannot hold modifier key info, since Ctrl or Meta is used to invoke them and Shift and Alt are used to generate different characters from the keyboard (e.g. **CSI 97= SP W** (Ctrl-a) and **CSI 65= SP W** (Ctrl-A, i.e. Ctrl-Shift-a) are different).

- **Meta/Command/Windows keys**. Used as a modifier key for arrow keys and some other keys. Used to make "character-function-keys" from printable character keys. We propose to extend the FNK (**CSI** *n* **SP W**) control sequences with allowing negative values (in decimal) whose absolute value refers to a character code (Unicode/ISO/IEC 10646 character code).

- **Menu key**. For the purposes of ECMA-48, if available for that: the same as Meta/Command/Windows keys. May open contextual menu otherwise.

- **Caps lock key** (and indicator light). When Caps lock is enabled, uppercase and lowercase are "swapped" on level 1/level 2, and on level 3/level 4. For keys that are not for cased letters, this lock is a no-op. We here propose to send **CSI 1=:1 SP W** when enabling Caps Lock, and sending **CSI 1=:0 SP W** when releasing Caps Lock, just as info to the target application (the actual "swapping" is local to the keyboard driver). We also propose that the application can send **CSI 1=:0 SP W** to disable the Caps Lock. (It is technically possible to use this key to instead switch between scripts, e.g. switch between a Greek keyboard layout and a Latin keyboard layout. But that is not commonplace today. A “language” keyboard switch is used instead. The latter can be supported via ECMA-48 style DSC control sequence, though details are not proposed here.)

- **Num lock key** (and indicator light). This is a slightly problematic “lock”. It is not available on “all” modern keyboard, and when available, it is not entirely clear what it does. While NumLock on turns “some” keys into a numeric keypad, exactly which keys are “turned” may vary between different kinds of keyboards. Since the digits (ASCII or local script) are usually available elsewhere on the keyboard, so num lock is rarely needed (unless one types a lot of numbers…).

- **Other lock keys** (on certain keyboards, like Japanese ones). Some keyboards may have more locking keys. One example is Japanese keyboards that may have locking keys for switching between Latin letters, normal width or wide, Hiragana, Katakana, and (converting last few characters to) Kanji. Korean and Chinese keyboards may have similar locking keys adapted. Other keyboard for “non-Latin” scripts may have similar locks, though switching between “local script” and Latin script (often limited to ASCII) may be done by switching “language layout” of the keyboard. (At this time, we do not propose ECMA-48 controls for this, but DCS strings may be used.)

- **Fn shift key**. Some modern keyboard have this key, since that (otherwise very rarely used) F1..F12 keys are overloaded with (more common) function keys for volume, brightness, etc. The Fn key is interpreted purely locally, no modifier key info.

- **F1**..**F12 keys**. xterm does have control sequences for these (now rarely used) keys. Unfortunately, xterm uses private use control sequences for them. But ECMA-48, fifth edition, already has standard control sequences for these keys: **CSI** *n* **SP W** (with positive *n*; here extended to be able to carry modifier key info **CSI** *n*;*m* **SP W**). They have almost universally, in use, been replaced by the character function keys (Ctrl-<char>, or Meta-<char>). So we propose to extend the FNK – FUNCTION KEY (**CSI** *n* **SP W**) control sequences with allowing negative values (in decimal) whose absolute value refers to a character code (Unicode/ISO/IEC 10646 character code). (Some of the negative values are used to refer to lock key functionality.)

- **Space key**. ……………………..SP, IDSP (only for CJK), NBSP (Shift-Sp), NNBSP (Alt-Sp),

- **Arrow keys**. The arrow keys are deceptively simple, up, down, left and right. That (seemingly) correspond to CUU, CUD, CUB, CUF. And they do. But… They can be modified with modifier keys. We here follow xterm (and its many derivatives) in extending the control sequences for these keys with modifier key information as an optional second argument. This is to allow following the use of arrow keys in windows environment. *Ctrl-LeftArrow* moves the cursor to the preceding word boundary (what constitutes a word boundary is implementation defined). *Shift-LeftArrow* moves the second text selection boundary to the left, *Shift-Ctrl-LeftArrow*, moves the second text selection boundary to the preceding word boundary. Likewise for the RightArrow key. By sending the modifier key information, ECMA-48 based applications can follow this way of interpreting arrow keys with modifiers.

- Misc. keys. Keyboards sometimes have a few other keys, like “print screen” and “eject”. At this time, we do not propose ECMA-48 control sequences for them (though values of *n* larger than **200** for **CSI** *n* **SP W** could be used).

- Speaker volume up/down/mute/unmute (&indicator light for mute); brightness up/down; microphone mute/unmute (&indicator light for mute); normally: local, not sent to “data component”); (out of scope: send these TO the “display component”, interrogation re. these)

5.2 Function keys – FNK (FUNCTION KEY) (extended)

Function keys are common on modern keyboards, usually (now) F1…F12. But they are rarely used as they were originally intended. So now these keys tend to be overloaded with controls for sound, brightness, etc. But to get the “original” function key, there is sometimes an “fn” shift key. xterm has support for function keys (i.e.; sending control sequences for them). Unfortunately, xterm does not use the standard control sequences, even though they have been in ECMA-48 “forever”. One should use the standard ones, and not any private use control sequence:

- **CSI 1**[;*m*] **SP W** F1
- **CSI 2**[;*m*] **SP W** F2

  …
- **CSI 12**[;*m*] **SP W** F12

  …
- **CSI 101**[;*m*] **SP W** PF1
- **CSI 102**[;*m*] **SP W** PF2

  …

5.3 Keyboard state locks (FNK extended)

Between **1=** and **30=** (note, negative values) some “special” functions are allocated. Here only =10…=12 are specified. Other values are reserved for future use; not all need be “lock” function keys.

- **CSI 1=:1 SP W** “Caps lock engage(d)” (just informative when sent to remote system),
  CSI 1=:0 SP W “Caps lock disengage(d)” (just informative when sent to remote system; can also be sent from remote to local system and then it is a command turning off the caps lock). Also **ESC c** should perform this.

- **CSI 2=:1 SP W** “Num lock engage(d)” (just informative when sent to remote system),
  CSI 2=:0 SP W “Num lock disengage(d)” (just informative when sent to remote system; can also be sent from remote to local system and is then a command turning off the num lock). Also **ESC c** should perform this. Some keyboards that do not have a separate numeric keypad section may simulate a numeric keypad “on top of” part of the letter keys section. However, it is (keyboard) implementation defined exactly which letters/punctuation keys are overlaid.

- **CSI 3=:2 SP W** “Katakana lock engage(d)” (just informative when sent to remote system, only available for Japanese keyboards)
  CSI 3=:1 SP W “Hiragana lock engage(d)” (just informative when sent to remote system, only available for Japanese keyboards)
  CSI 3=:0 SP W “Rōmaji lock engage(d)” (just informative when sent to remote system, only available for Japanese keyboards)

These can also be sent from remote to local system and is then a command to switch keyboard (driver) mode.

- …
- **CSI 30=:1 SP W** FN lock engage(d). Some keyboards have an “fn” key that can work both as a “shift-like” key, and like a “shift-lock-like” key. This control sequence signifies FN lock engage (command from remote to local) or engaged (information from local to remote).
  CSI 30=:0 SP W FN lock disengage(d). Some keyboards have an “fn” key that can work both as a “shift-like” key, and like a “shift-lock-like” key. This control sequence signifies FN lock disengage (command from remote to local) or disengaged (information from local to remote).

~~Also **ESC c** (RESET TO INITIAL STATE), if implemented, should perform these resets (disengage; for **CSI 3=:2 SP W** the default is implementation defined) of keyboard state, but **ESC c** resets much more.~~

The lock engaged/disengaged sets of control sequences should also be included in the response to **CSI 6n**/**CSI 6!n**, since they inform about the keyboard state. (We here do not propose a “language” keyboard layout information or command; though one could be made using a **DCS…ST** control string.) Note that only the application/tile that has the keyboard focus will get these engage/disengage control sequences when the key is pressed.

Note that “scroll lock” is not a keyboard lock state, so cannot have lock/unlock control sequences.

The DISABLE/ENABLE MANUAL INPUT escape sequences, intended for locking/unlocking keyboard (and, by extension, mouse) input, are very strongly deprecated, and must not be implemented.

5.4 Speaker volume, screen brightness, microphone and camera states

The presentation component nowadays can control the screen brightness (incl. “muted”/”not muted”), sound volume (incl. “muted”/”not muted”) as well as microphone “muted”/”not muted” and camera via software commands. Those settings may even be allowed to be controlled “remotely” (i.e., what ECMA-48 calls the “data component”). ECMA-48 is often geared towards text based applications, and not so much video (while playing, “screen lock” or “screen mute” is normally disabled by commands from the “data component” to the “presentation component”), sound, or “call” type of applications (in a conference call, one may be able to remotely mute other participants…), but certain things are certainly still applicable when it comes to control sequences also for such applications, in addition to that mouse and keyboard input may be used.

At this point we do not suggest precise control sequences for these functions. However, we may do so in a future revised proposal.

5.5 Character function keys (FNK extended)

Character reference function keys are mapped to negative values strictly less than −31 (**31=**). The character references are decimal, with no leading zeros, and refers to a character normally generated by the keyboard, the character that has the code that is the negation of the parameter.

- **CSI** *n*= **SP W** (Positive decimal integer value for *n*, strictly greater than 31, no leading zeroes; note that *n*= thus refers to a negative integer value.) Control sequence for when the shortcut key (ctrl or command/windows key) is pressed plus what would otherwise generate a character from the keyboard. *n* is the *decimal* number (no leading zeroes) for the character that would have been generated if the shortcut key had not also been pressed. For a terminal emulator application, these letter-function keys strokes should normally (i.e., by default) be handled locally. Like Ctrl-c for “copy” (locally) and Ctrl-v for “paste” (locally).

In certain situations, in terminal emulators (not other applications), one need to be able to generate (some) C0 control codes from the keyboard. Like traditional Ctrl-c (for ETX) for making the (“cooked”) TTY driver send a SIGINT to the program that has that TTY as stdin (Unix/Linux); which is an abuse of ETX, but that cannot be rectified. Just typing Ctrl+c for that, which is traditional, causes problems for nearly all modern users, who are used to Ctrl+c for “copy selected text” (locally to a copy-paste buffer). Then instead causing the program to be stopped is annoying and may cause a lot of extra work.

Instead, we propose to use Ctrl-Ctrl-c (and similar for other C0 controls), i.e., double-click the Ctrl key and hold down and then “c” key for sending an ETX to the remote side (likely causing the target program to terminate, unless the SIGINT is caught). But this is for terminal emulator applications only.

5.6 Remote/local switch for character function keys (FNK extended)

For terminal emulators, some character function keys may be interpreted on the presentation component rather than sent to the data component. In addition, most programs on the “data component” side are not made to at all handle character function keys. For these reasons, the “data component” program that handles (some) character function keys need in that situation tell the “presentation component” that it will interpret certain character function keys, and thus want them forwarded to the “data component” and not be interpreted on the “presentation component”. There is also a reset for this, but note that **ESC c** (

The “function keys” have basically completely been replaced (in many systems) by character function keys. Essentially because they are (semi)mnemonic: e.g., Ctrl-s (or Meta-s) for “save”, Ctrl-S for “save all”, Ctrl-c for “copy (to copy-paste buffer)”, … But in existing terminal emulators, these may be interpreted “locally”, i.e., by the terminal emulator itself.

CSI 31=…SP W is used to allow remote side to “take over” the handling of “letter function keys”, which are normally handled locally. It is implementation defined if similar “take over” controls are needed also for strictly positive FNK arguments.

- **CSI 31=**[**;**<*semicolon separated list of negative FNK code values requested to be handled remotely*>] **SP W** Remote system request that the character function keys are to be handled by the remote system. Local system sends **CSI 31=**[**;**<*semicolon separated list of negative FNK code values that will be sent to the remote side*>] **SP W** as acknowledgement. Hereafter character function keys send character function key control sequences to the remote system, and they mostly not handled locally. **CSI 31= SP W** Initial/default, empty list of negative values. Some character function keys (may be settable in preferences) can still be forced to be handled locally (e.g. ctrl—and ctrl-+). This control sequence is tailored for terminal emulators, where some character function keys are often handled by the terminal emulator itself (locally), and so far few “remote” programs at all process character function keys. A reset to handle character function keys locally is done by **CSI 31= SP W** or **ESC c** (but note that the latter resets much more).

6 Mouse/touchpad/touchscreen input (new, but cmp. xterm)

The use of “mice” (some with a scroll wheel) to do interactions on computers (with GUI display) postdates ECMA-48 5th edition. (“Rollerballs” are here handled as equivalent to mice.) But xterm does have extensions to handle mouse actions (in addition to keyboard actions, like arrow keys). Thereafter we got multitouch touchpads and then touchscreens, and we also got modifier key modifications to mouse actions (e.g., ctrl-click). All of these later options and extensions are not handled by the xterm handling of mouse input, and in addition it is not done in a way that is ECMA-48 compatible; it’s just … wrong… However, it is perfectly possible to not only be ECMA-48 compatible, but also be able to handle touchpads and touchscreens in addition to mouse input.

Modifier key codes (*m* below) here are the same as for text edit movement control sequences (see above), and FNK control sequences (also above). *n* is number of click for click actions: *n*=**1** for single click, *n*=**2** for double-click, …

Coordinates are in *em* of the default font size and may be fractional or negative (or both). Origin is upper left corner of the tile, not counting any kind of frame (tab, window). [Coordinates are in pixels from the upper left corner of the tile with positive rightwards and downwards, in the screen resolution of the upper left corner of the tile, not counting any kind of frame (from tab or window). Full screen(s) mode for a tile does not have any frame but must be rectangular.]

- **CSI 0**[**;?;**<*semicolon separated list of mouse/touchpad/touchscreen code values requested to be handled remotely*>]**!@** (A **1** in the list can have an angle resolution modifier: **1:10** 10 degree resolution (when reporting to the remote system), **1:30** 30 degree resolution, **1:45** 45 degree resolution, **1:90** 90 degree resolution. Default is one degree resolution.) Can be sent from remote system to tell which trackpad/mouse/touchscreen actions to send to the remote system and not handle locally. However, rotating a handheld device can still cause window size change even if the rotation control sequence is sent to the remote system. **CSI 0;***r*[**;***x***;***y*][**;?;**<*semicolon separated list of mouse/touchpad/touchscreen code values that will be sent to the remote system*>]**!@**is reported back to acknowledge the change, the list of codes may be a subset of those requested. The *x***;***y* is used when acknowledging the changeover (same control sequence apart from the *x*;*y*) indicate the pointer location, in virtual pixel indices (to compensate for different screen resolutions; implementation defined; the coordinate origin is at the virtual upper left corner of the tile, positive is rightwards (*x*) resp. downwards (*y*)). No *x***;***y* position is usually reported back for handheld devices since they usually do not have a pointer position. *r* is virtual resolution pixels per virtual cm; how this depends on the resolutions of the multiple screens is implementation defined; how this is mapped for very large screens (“jumbotrons”) is implementation defined (but a virtual cm may then actually be decimetres). (Virtual resolution does not change in case of temporary zoom (commonly available as local handling on touchscreens).) A **CSI 0!@** requests to restore the [terminal emulator] default of not reporting such events to the remote system; same is sent back as acknowledgement. A reset to handle mouse/etc. input locally is done by **CSI 0!@** or **ESC c** (but note that the latter resets much more). This control sequence is tailored for terminal emulators, where mouse/etc. input is often handled by the terminal emulator itself (locally), and so far few “remote” programs at all process mouse/etc. input.

- **CSI 1;***angle***!@** Absolute screen rotation (for handheld screens). Angle is rotation in degrees, positive is anticlockwise, 0 degrees is screen held “upright”, “portait”. Postfix **=** for negative values, an use **?** for decimal marker. Angle range **180?0=** to **180?0** (no leading zeroes). Can be sent “spontaneously” when there is a sufficiently large (according to requested resolution) rotation, or “on demand” requested by **CSI 1!@**. The result is rounded to the requested angle resolution.

- **CSI 2;***n***;***x***;***y***;***x′***;***y′*[**;***m*]**!@** Non-grab movement. Coordinate transformation can be set in preferences for touchpads and mice. Actually does a relative move of the pointer; so it is not touchpad coordinates (when a touchpad is used), it is pointer coordinates. This kind of movement is not available for touchscreens. Multi-point non-grab movement (can translate to scroll or zoom), *n* is number of points (track each point??).

- **CSI 3;***n***;***x***;***y*[**;***m*[**;***f*]]**!@** Click and hold. *n* is number of clicks before the hold of one second without any movement.

- **CSI 4;***n***;***x***;***y***;***x′***;***y′*[**;***m*[**;***f*]]**!@** Grab movement. Coordinate transformation can be set in preferences for touchpads and mice, but is fixed for touchscreens. Actually does a relative move of the pointer; so it is not touchpad coordinates, it is pointer coordinates, as opposed to touchscreens. Postfix = for negative values, ? for decimal marker. Starts movement grab if not already active. *f* is “grab force”, implementation defined. [If not grabbing an existing text selection: Will set start or extend text position/selection to nearest available text insertion position (bidi……).] *n*: 1 for single-click grab, 2 for double-click grab.

- **CSI 5;***x1***;***y1***;***x1′***;***y1′***;***x2***;***y2***;***x2′***;***y2′***!@** Move/zoom/rotate grab movement. *x1***;***y1***;***x1′***;***y1′* is for first point movement, *x2***;***y2***;***x2′***;***y2′* is for second point movement. Starts move/zoom/rotate grab if not already active. Can be extended for more moving points (3 or 4), but then does not grab. Coordinate transformation can be set in preferences for touchpads (and actually moves/zooms/rotates relative to pointer position, if at all implemented) but is fixed for touch screens. This kind of movement is not available for mice. Postfix = for negative values, ? for decimal marker, though unit is display pixels.

- **CSI 6**[**;***x1***;***y1*[**;***x2***;***y2*]][**;***m*]**!@** Release grab, both for grab movement (one point), and move/zoom/rotate grab movement (two points).
  CSI 6!@ from the remote side can be used to force a release. Force release of grab (e.g. when remote is side gets a **CSI 9!@**). Turn **CSI 4;***n***;***x***;***y***;***x′***;***y′*[**;***m*[**;***f*]]**!@** into **CSI 3;***n***;***x***;***y***;***x′***;***y′*[**;***m*]**!@**, releasing (not dropping at new location) the object (like a selected text). Can be sent from remote side. Response is **CSI 6**[**;***x1***;***y1*[**;***x2***;***y2*]][**;***m*]**!@** as confirmation. Note that this is a UI feature, even though it may appear as a security feature. (There is no validation or guarantee and there may be other means of getting objects out, such as cut-and-paste.)

- **CSI 7;***btn***;***n***;***x***;***y*[**;***m*]**!@** Button click (down&release without movement). *btn* is 0 for trackpad click, 1…j for other buttons. *x,y* relative to upper left corner of window (with positive downwards). ~~They are in percent of height and width of the window~~. n is 1 for single-click, 2 for double-click, … (Click-grouping time delta setting?) [Will move active position in text to the closest available text edit position.

- **CSI 8;***z***!@** Mouse scroll-wheel rotation. Postfix = for negative values (rotating top of wheel towards user), ? for decimal marker. Scroll wheel scaling can be set in preferences. Some track pad implementations simulate scroll wheel on the right side. (xterm maps scroll wheel movement directly to scroll control sequences…?) m? unit?

- **CSI 9;*g*!@** Sent to the target data component when the pointer location is outside of the display of the corresponding display component. *g*=**0**: non-grab movement in progress, *g*=**1**: grab movement in progress. Should be sent when the pointer location exits, but should be sent sparingly otherwise. For touchscreens this will be sent when there is an "!@" action in the display of another display component.

- **CSI 10;*g*!@** Sent to the target data component when the pointer location is inside the display of the corresponding display component. *g*=**0**: non-grab movement in progress, *g*=**1**: grab movement in progress. Should be sent when the pointer location enters, but should be sent sparingly otherwise. For touchscreens this will be sent just before there is an "!@" action in the display of this display component.

- **ESC ´** DMI - DISABLE MANUAL INPUT. Sent to the target data component when it does not have the keyboard focus. This escape sequence is likely in origin intended to be sent from the data component to the display component (with keyboard…), but here we use it in the other direction to inform the data component that it does not have the keyboard focus. Should be sent when to the data component when the keyboard focus is lost, but should be sent sparingly otherwise. (Sending this escape sequence to the display component is out of scope for this proposed update.)

- **ESC b** EMI - ENABLE MANUAL INPUT. Sent to the target data component when it has the keyboard focus. This escape sequence is likely in origin intended to be sent from the data component to the display component (with keyboard…), but here we use it in the other direction to inform the data component that it has the keyboard focus. Should be sent when to the data component when the keyboard focus is gained, but should be sent sparingly otherwise. (Sending this escape sequence to the display component is out of scope for this proposed update.)

7 Tile and window manipulation

Most other parts of ECMA-48 can be used for various implementations. But the tile and window manipulation operations are specially targeted at terminal emulator implementations. Other implementations, like text editors, are expected to completely ignore (literally) these control sequences.

7.1 Tile and window manipulation (new, but compare xterm)

The private use control sequences **CSI?1049h** and **CSI?1049l** in xterm switch to and from an "alternate buffer" (which we here call overlay tile). But they are private used, so cannot be standardised. For each tile stack, it is only the top tile that is displayed, and all tiles in a tile stack have the same size.

Some of the tile property change operations need a string argument, and that cannot be provided by **CSI**…**!T**. We need that for the set window titles operations. These are associated with tiles and overrides the default window title(s) when set. The title(s) used for a (terminal emulator) window is the one(s) set for the tile that has the keyboard focus or would get the keyboard focus if the window is deiconified and given the keyboard focus. The title of a tab is that of the tile within the tab that has or would get the keyboard focus if the tab gets the keyboard focus. A tile may have its own title bar.

The window manipulation control sequences can go in both directions:

- From local system (that actually displays the window) to the remote system, informing the remote system about a window related change, a move, size change, or iconify/deiconify. The remote system may respond by redrawing the content of a fixed type overlay tile (tiles that are not of fixed type may have a local buffer that may be larger than the tile, and any redrawing is done locally).
- ~~From the remote system to the local system, letting the remote system change the window in some way (size, position, (de)iconification, …) Only on user instruction should they at all be interpreted. That is because these controls may affect not only the content and behaviour of a single "tile stack" (for one terminal emulator instance), but also other "tile stacks" and indeed also other windows. (In current xterm, the corresponding private use controls can and should by default be be disabled using the *allowWindowOps* resource. One can use a handier, for the individual user, way of disabling(default)/enabling these controls.)~~

These control sequences correspond to the private use control sequences **CSI**…**t** in xterm. But private use control sequences of course cannot be standardised. In addition, it is not quite right for multiple screens, nor for multiple (terminal emulator) tiles, either in tabs and/or laid out in a grid.

All of the window manipulating control sequences from the remote side should, also when allowed/interpreted, be used with great caution, since they can be very disruptive for the user. They affect not only one tile (stack), but also other tiles and also the window itself and other windows.

- **ESC c** RIS - RESET TO INITIAL STATE. This resets the tile stack to its initial state, including that there a no overlay tiles. However, the default point size and default line spacing is not changed. (So one can set the default point size for the (base) tile (some terminal emulators use Ctrl-+ and Ctrl-- for that), and the thus set point size is not changed by **ESC c**. (The Unix/Linux command "*reset*" sends **ESC c** to the terminal emulator.)

- **ESC e** Push "traditional terminal emulator" tile overlay. Overlays may be limited to have no scroll buffer (can still be resized or get an em or line spacing change), and limited to default size (can be changed via pseudo-"zoom"; as a kind of resize in terms of number of lines and columns, not pixels) and default fixed width font in order to allow for programs such as emacs, vim, less, …). ESC f pushes a "cells" tile. Like a traditional terminal emulator (almost), no bidi, possibly only horizontal lines, only the default fixed width font (but allow for bold and italic), only the default font size (though that may change per tile locally, not via control sequences), only the default line spacing, no subscript/superscript, no local scrolling (only scrolling via scrolling control sequences to the remote side, and remote side redrawing), no images, no vector/plotter graphics, no math expression layouts in the sense of *A true plain text format for math expressions (and its XML compatible equivalent format)* nor as TeX math expressions.   ? This is called *Alternate Screen Buffer* (**CSI ?1049h**) in xterm and is limited to a stack depth of 1. All tiles in a tile stack has the same size, but need not have the same default font size nor the same default line spacing. The latter are however (initially) inherited from the preceding tile in the stack.

- **ESC f** Push a "freer layout" tile (if "freer layout" tiles are implemented, otherwise push a traditional terminal emulator tile). This overlay tile type is like the "lowest" (the non-overlay) tile. It may permit proportional fonts, bidi(?), math layout, line spacing changes (which are not changing the default), font size changes (which are not changing the default), and local scrolling (i.e. having a local "buffer" larger than the tile size) are all permitted. Other limitation are not fixed, but are implementation defined.   ?

- **ESC g** Pop the top overlay tile. No-op if no overlay tiles, i.e., the lowest tile (non-overlay) cannot be popped. (In xterm: **CSI ?1049l**.)

- **ESC h** Pop all tile overlays. This can also be done via **ESC c**, which do more resets than this.

- **ESC i** Iconify the tile or even parent window of the tile. Parent window will not be iconified if there are other tiles in the window that are not iconified. Need not be interpreted even if there is only one tile in the window. Can be sent to remote side as info that the tile is iconified. (**CSI 2t** in xterm.)

- **ESC j** Deiconify/ied parent window. When from remote side (and enabled from remote): De-iconify the parent window (to its latest position, size and tile layout) of the tile that got this control sequence. Put that window on the top of all windows and put the parent tab of the tile as the active tab in the window. Can be sent to remote side as info that the tile and its parent window are deiconified (it may still be under other windows). (**CSI 1t** in xterm.)

  (((How subscribe to the above? They should not be sent to "unaware" applications.)))

- **CSI 0**[;<*semicolon separated list of tile/window manipulation codes*>]**!T** Tell remote system which tile/window manipulation operations are allowed. If more than 3 window manipulation operations are received within 3 minutes, then all remote window manipulation is turned off, and there is no way for the remote system to automatically turn it on again. **CSI 0!T** (default) tells the remote system that no tile/window manipulation operations sent from the remote side will be interpreted. Sent from remote system to local system to "preadvice" which window/tile manipulations the program would "like" to do. Also used as response, telling which of those operation will be executed if received. All other tile/window operations will be ignored. The default response should be **CSI 0!T**, i.e. no such operation from the remote side will be executed. However, **CSI 0**[;<*semicolon separated list of tile/window manipulation codes*>]**!T** and **CSI 1!T** are always permitted, and should be responded to.

- **CSI 1!T** Ask for tile (stack) size info. Replied to with **CSI 1;***dx***;***dy***;***em***;***ls*[;*x1***;***y1***;***x2***;***y2*]**!T**. The optional part of the response is omitted if local side has not given permission for remote side to issue **CSI 1;***dx***;***dy***;***em***;***ls*[;*x1***;***y1***;***x2***;***y2*]**!T** control sequences. This response is given even if there is no change in size; from the requesting remote side the size may be unknown or need refresh.

- **CSI 1;**[*dx*]**;**[*dy*]**;***em***;***ls*[;*x1***;***y1***;***x2***;***y2*]**!T** Size change of tile (normally sent to remote system; may be sent to local system). *dx* is the resulting width of the tile in pixels, *dy* is the resulting height of the tile in pixels, *em* is the size in pixels of the em of the default font size. *x1***;***y1***;***x2***;***y2* (in pixels) are the upper left corner and lower right corner of the tile relative to the upper left corner of the "tiles area" of the parent window. These coordinates may be negative. If interpreted when this control sequence is sent from the remote system to the local one, *x1*;*y1*;*x2*;*y2* are ignored and should be omitted; other tiles may be moved or change in size (implementation defined), and the size of the parent window may be adjusted in size and position (implementation defined). If *x1*;*y1*;*x2*;*y2* are given *dx* and *dy* should be omitted (ignored, and no check that dx=x2-x1 or dy=y2-y1). Change is done even if parent window is iconified. *ls* is line spacing in em, default is 1.2 em (given as **1?2**).

- **CSI 2;***x1*;*y1*;*x2*;*y2***!T** Size and position of a tile's parent window (changes it even if the window is iconified). Coordinates are relative to main screen's upper left corner. These coordinates may be negative. If interpreted when this control sequence is sent from the remote system to the local one, also other tiles may be moved or change in size. Change is done even if parent window is iconified. When resizing/moving (if permitted) a window, the resize/move should be limited so that at least a part of the title bar is visible on at least one of the screens.

- **CSI 3;***dx*;*dy*;*em*;*ls***!T** Full-screen for tile (stack). When from local side When from remote side (and enabled from remote): If tile's window is iconified then de-iconify the window of the tile, and put window on top of other windows, make tab of tile the current tab. Make the tile fullscreen (at top of all windows) on all screens that the window is in. full screen for tile vs. full screen for window… Can be used when already in full screen if the default em is changed (some terminal emulators do that as "zoom"). (((Only vertically, only horizontally, …))) No-op if more than one tile in the window??

- **CSI 4;***dx*;*dy*;*em*;*ls*[;*x1*;*y1*;*x2*;*y2*]**!T** Exit(ed) (vertical/horizontal/both) full-screen for tile (stack), and back to deiconified window.

- **CSI 5;***n***!T** Set pointer icon for tile (in a tile stack, each tile has their own pointer icon, but only the top one is used). If *n* is **0**, then the pointer icon is invisible, other values are implementation. (Exceptionally this is basically intended for being sent from the data component to the display component, if ECMA-48 is used for that communication.)

- **CSI 6;***n***!T** Make the entire tile blink *n*, 1 <= *n* <= 5, times (by inverting the colours). Can be used as replacement for sounding the bell, which is both noisy and hard to tell where the "alert" is coming from. (A preference setting can (implicitly) convert BEL to **CSI 8;***n***!T**.) If this control sequence is received while blinking, the control sequence is ignored. Blinking must not block other processing. (Exceptionally this is basically intended for being sent from the data component to the display component, if ECMA-48 is used for that communication.)

The following three types of control strings are taken directly from xterm. The default strings here are implementation defined.

- **OSC 0;***<string>* **ST** Change icon name and window title property of the tile to <*string*>, replacing any previous settings for the tile, to be used to override the default strings for the tile. (Recall that the space after OSC and the space before ST is part of the notation, there should be no spaces there in the actual control string.)

- **OSC 1;**<*string*> **ST** Change icon name property of the tile to <*string*>, replacing any previous setting for the tile, to be used to override the default string for the tile.

- **OSC 2;**<*string*> **ST** Change window title property of the tile to <*string*>, replacing any previous setting for the tile, to be used to override the default string for the tile.

SGR style changes within these strings are (for now) not interpreted. However, an implementation may selectively interpret some style changes, as well as some math expressions. It depends on the target system what can be displayed in such title strings. By their very nature, the strings need be fairly short to fit as icon or window titles. Therefore, an implementation may cut these titles short, and even output the overflow to the ordinary output (the tile itself, instead of the title bar).

*Ps = 3 ; x ; y ⇒ **Move window to** [x, y].*

*Ps = 4 ; height ; width ⇒ **Resize the xterm window** to*
given height and width in pixels. Omitted parameters reuse
the current height or width. Zero parameters use the
display's height or width.

Window
*Ps = 5 ⇒ **Raise the xterm window** to the front of the stacking order.*
*Ps = 6 ⇒ **Lower the xterm window** to the bottom of the stacking order.*
*Ps = 7 ⇒ **Refresh the xterm window**. <←--*

Tile
~~*Ps = 8 ; height ; width ⇒ **Resize the text area** to given*~~
~~*height and width in characters. Omitted parameters reuse the*~~
~~*current height or width. Zero parameters use the display's*~~
~~*height or width.*~~

Window
Ps = 9 ; 0 ⇒ Restore maximized window.
Ps = 9 ; 1 ⇒ Maximize window (i.e., resize to screen size).
Ps = 9 ; 2 ⇒ Maximize window vertically.
Ps = 9 ; 3 ⇒ Maximize window horizontally.

Tile
*Ps = 1 0 ; 0 ⇒ **Undo full-screen mode**.*
*Ps = 1 0 ; 1 ⇒ **Change to full-screen**.*

*Ps = 1 1 ⇒ **Report xterm window state**.*
If the xterm window is non-iconified, it returns CSI 1 t .
If the xterm window is iconified, it returns CSI 2 t .

*Ps = 1 3 ⇒ **Report xterm window position.***
Note: X Toolkit positions can be negative, but the reported
values are unsigned, in the range 0-65535. Negative values
correspond to 32768-65535.
Result is CSI 3 ; x ; y t

*Ps = 1 3 ; 2 ⇒ **Report xterm text-area position**. Result is CSI 3;x; y t*

*Ps = 1 4 ⇒ **Report xterm text area size in pixels**.   Result is CSI 4;height;width t*

*Ps = 1 4 ; 2 ⇒ **Report xterm window size in pixels**.*

Normally xterm's window is larger than its text area, since it includes the frame (or decoration) applied by the window manager, as well as the area used by a scroll-bar.

Result is CSI 4 ; height ; width t

*Ps = 1 5 ⇒ **Report size of the screen in pixels**. Result is CSI 5 ; height ; width t*

Ps = 1 6 ⇒ Report xterm character cell size in pixels.   Result is CSI 6 ; height ; width t

~~*Ps = 1 8 ⇒ Report the size of the text area in characters. Result is CSI 8 ; height ; width t*~~

~~*Ps = 1 9 ⇒ Report the size of the screen in characters.   Result is CSI 9 ; height ; width t*~~

Ps = 2 0 ⇒ Report xterm window's icon label.   Result is OSC L label ST

Ps = 2 1 ⇒ Report xterm window's title.   Result is OSC l label ST

8 Cut and paste

Some common UI actions today require a “system global” (not quite, it is per user, or rather per login; the exact details are beyond the scope of this proposal) state. In particular, cut-and-paste which uses a “global” clip buffer (which in general can contain not only text, but also images and other items that can occur; here, however, we focus on text). Some ECMA-48 based applications (read terminal emulators and programs that are intended to be run interactively via a terminal emulator) already support mouse movements and clicks via implementation specific extensions to ECMA-48, and below we will propose ECMA-48 extensions to handle not only mice and other “pointer handling” devices, but also the common “ctrl-<letter>” commands handling the cut-and-paste buffer (often Ctrl-x, Ctrl-c, and Ctrl-v). It stands to reason that one would expect the “globality” of the cut-and-paste buffer to extend also between the local system and the remote system (in a terminal emulator), using an ECMA-48 based connection (in both directions, as is the case for terminal emulators).

Then the issue arises on how to synchronise the remote system (program) and the local system w.r.t. the copy-paste buffer. Recall that we here focus on text, not expecting to be able to much anything else in an ECMA-48 context (but extensions are possible). The synchronisation can be done via ECMA-48 based messages sent between the local and the remote system. This requires cooperating remote program and local program (terminal emulator). We here propose to use the OSC (Operating System Command) feature of ECMA-48 (which is completely open-ended in the fifth edition of ECMA-48). This can then be used to, invisibly to the user, communicate the (text) content of the cut-paste buffer between remote and local system.

We propose the **OSC cp-buffer1=**<*contents of cut&paste buffer 1*> **ST** format for sending the (text) content of the cut-paste buffer to the other (cooperating) system. When a copy-paste buffer is changed remotely, this is sent to the local system. When a copy-paste-buffer is changed locally, this is sent to the remote system. Sending this from the local side may be limited to when the remote side has enabled either mouse control sequences or character based function keys (see below).

The issue of styling conversion in connection with cut-and-paste is discussed in section 13 of *Text styling control sequences – modernised ECMA-48 (ISO/IEC 6429) text styling* (2020).

9 Conclusion

10 REFERENCES

[ECMA-48]	*Control Functions for Coded Character Sets*, 5th ed., June 1991, https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-048.pdf. (Earlier editions as well.)
[ISO/IEC 6429:1992]	Information technology – *Control functions for coded character sets*, 3rd ed. Same as [ECMA-48] 5th ed.
[ISO/IEC 10646]	Information technology – *Universal Coded Character Set* (UCS), https://standards.iso.org/ittf/PubliclyAvailableStandards/c069119_ISO_IEC_10646_2017.zip. Same coded characters as in [Unicode].
[Math controls]	Kent Karlsson, *A true plain text format for math expressions (and its XML compatible equivalent format)*. 2022.
[Styling upd. ECMA-48]	Kent Karlsson. *Text styling control sequences – modernised ECMA-48 (ISO/IEC 6429) text styling*. 2022.
[Unicode]	*The Unicode Standard*, http://www.unicode.org/versions/latest/, web site https://home.unicode.org/, http://unicode.org/main.html, …, 2017, …
[XTerm control seq.]	*XTerm Control Sequences*, https://invisible-island.net/xterm/ctlseqs/ctlseqs.html, 2019.