Text styling control sequences – modernised ECMA-48 (ISO/IEC 6429) text styling

Kent Karlsson
Stockholm, Sweden
2025-11-10

1 Introduction

There are several higher-level mark-up techniques for styling text. Web pages and many smartphone apps use HTML combined with CSS (either embedded or by reference). RTF (Rich Text Format, an old format for MS Word; replaced by *Office Open XML*) and TEX (and LaTeX, etc.) are other examples of markup. There are several more. The origin for HTML, SGML, is also still used.

Some significantly simpler markup languages are called “markdown”, but “markdown” still uses (only) ‘printable’ characters (thus easy to enter by keyboard) characters you would normally want to display “as-is”. So their styling control cannot be invisible in plain text, only when interpreted. To make matters worse, there is usually no quoting mechanism in “markdown”. “Markdown” can have surprising effects, like turning −nnn− unexpectedly into text with strike-through: ~~nnn~~ in some of the variants; and actual * in the text unexpectedly turns part of the text to bold. This is a problem for all “marking” languages (“markdown”, HTML, any XML-based markup, as well as TeX/LaTeX and old troff).

Sometimes a somewhat more light-weight mark-up (but not necessarily one lacking power) is desired, such as mark-down, but *with the added requirement* that it be “invisible” at the “plain” text level (and better at avoiding surprising unintended formatting and other pitfalls of many higher-level mark-up schemes). One could try to design one from scratch. But… there is already one such scheme, indeed a well-established one, and in common use, namely ECMA-48 *control sequences* that control text style. While the use of ECMA-48 control sequences may seem old-fashioned, it has several advantages, and there is no need to consider it outdated. Indeed, many of the problems with “printable” characters only-based marking schemes is avoided in the ECMA-48 scheme. Here is a list of some of the advantages of ECMA-48:

1. ECMA-48 is a well-established standard, since several decades by now. The first edition dates from September 1976, nearly 50 years ago. The latest edition is from 1991, more than 30 years ago. It does need some updating, for Unicode, for modern display capabilities, modern document display and edit conventions, additional capabilities common in “higher level” schemes for styling text, modern font capabilities, etc.
2. The ECMA-48 control sequences have the advantage that they can be handled (interpreted or ignored (but displayed, see below)) at the text level, no need for a parser at *higher* level. (It would still be considered a “higher level protocol” in the Unicode sense.) That is, due to the use of a (lead) control code, the mark-up does not consist solely of printable characters, like in so-called mark-downs (like that used in Wikipedia and elsewhere) or HTML, RTF and the like. The syntax is such that ECMA-48 control sequences, following the standard, can be parsed easily and “blindly” (the parsing step does not need to know if the control sequence is interpreted; note though, that in a later step, most *interpreted* control sequences/characters themselves are not rendered,

but have some effect, like changing the colour of the text; *uninterpreted* control sequences/characters **shall** be rendered in some way, not be invisible (that would be a misunderstanding of the ECMA-48 standard), and there is no need for any secondary encoding of some (in plain text) "non-control" characters (like "&lt;" for the '<' symbol in HTML). The reason for that is that the ("control") characters used in the ECMA-48 markup are *reserved* for *that* purpose, which is not the case for HTML, markdown, etc.

3. ECMA-48 control sequences are still in very common use. They were particularly popular for terminals, that were later replaced by terminal emulators, which is what is used today. The latter have since been developed beyond the limits of any once-existing physical terminal model. The continued popularity of ECMA-48 control sequences is partly for historical reasons, and but mostly due to point 2 (thus, HTML, RTF, mark-down etc. are forever out of the question for terminal emulators). Those who use terminal emulators will encounter ECMA-48 styled text in just about every use instance. For example, it is used for the Unix "`man`" command (print manual pages on the terminal); underline and bold occurs through-out those pages. It is also used by many other commands for colouring (e.g., `grep` in modern Linux can mark matching substrings in bold red), as well as being popular for use in prompts, often embedding certain status data as styled text in the prompt. Many other programs that output to a terminal also use ECMA-48 styling of the output. Even though ECMA-48 is "old" it is very much still in active use and there is no end in sight.
4. The SGR (SELECT GRAPHIC RENDITION) control sequences (including the extensions proposed below) and HTSA2, SPD2, PTX (also described below) are well interpretable, for now in principle, in modern text editors, including that SGR may be used to do styling inside math expressions. Just like formats such as RTF, HTML (usually, very partially), etc., are supported by many modern text editors. Likewise, if desired, charset declaration, as well as images and graphics, see hints below (but is *out of scope* for the *here* proposed additions), based on ECMA-48 are well supportable in text editors. Thus, one could have a special file format (with a suitable extension such as "txtf": *file*.**txtf**) for text styled with these ECMA-48 control sequences. Such a format would exclude interpretation of cursor movements (e.g. from arrow keys or mouse), terminal screen editing, or other control sequences that are not aimed at text formatting.

This is the closest one can get to "plain text formatting" (since control characters mark the "markup") *while using an existing standard*. (In "higher level markup", what is traditionally called "printable" characters act as control sequences per the syntax of the markup. That goes for HTML, RTF, markdown, etc. To be nit-picking, also "control" characters are also printable, see below, but more as an error indication when they are not interpreted.)

Some more items of interest w.r.t. ECMA-48:

1. ECMA-48 styling is popular to use for bold, underlining, italics as well as foreground and background colours. There is already support in the standard for much more, even though some need further extensions to be useful. In addition, some features should be moved to SGR (from other control sequences), to improve the design. Many implementations already support extensions w.r.t. colours and underlining (e.g., wavy underlines and colour of underlines separate from colour of the text). ECMA-48 also has support for CJK style of emphasis (similar to underlining).
2. A variety of PTX (PARALLEL TEXTS) is used for so-called "ruby" text, which is a particular kind of clarification text written above/beside the actual (usually Japanese or Chinese)

text. This is a feature only relatively lately added to HTML. But ruby text is supported, standards-wise, by ECMA-48 since more than 30 years, long before it was added to HTML.

3. ECMA-48 may appear not to have any mechanisms for specifying tables (though tab stops can be set, preferably via (resurrected) HTSA (HTSA2, CHARACTER TABULATION SET ABSOLUTE 2)). But ECMA-48 has another variety of PTX which, with some minor extensions (see below), can be used to make table rows, and hence true tables. Together with block frames (proposed below), cell frames and cell background colours can be supported. PTX and SDS/SRS (the latter two are for bidi control, similar to some of the nesting Unicode bidi controls, and indeed we will consider equivalent to certain Unicode) are the only control sequence from ECMA-48 5th edition that allows for nesting (so one can have tables in tables, or "ruby" text in tables (though "ruby" in "ruby" or tables inside ruby may be less meaningful).
4. Headings and item lists are supported as *direct* formatting (and direct numbering) only (in particular: size and bold or bold italic, as well as tab stops and HTJ). There are no "meta-formatting" controls by giving some kind of "element type", and then elsewhere define the formatting for instances of that "element type" (cmp. CSS, which is a prime example of that approach, but exists also in, e.g., TeX) for this or anything else, like paragraphs or table cells. That may in a modern context be seen as a major flaw. But it is a basic design feature of ECMA-48 formatting, and therefore cannot be changed, or at least quite hard to change. We will therefore not introduce any "meta-formatting" controls here, though not prohibited for a future update.
5. Images and vector graphics have no apparent support in ECMA-48. But non-support for such features is not quite true. There are extension mechanisms that are "private use" or "for future standardisation" that can be used for this. In particular, APC (APPLICATION PROGRAM COMMAND), which takes a string parameter that can encode an image (in base-64 or use a link) or a kind of vector graphics (plotter commands or even SVG). (Some more hints below, but details are out of scope for this proposal.)
6. Hyperlinks were barely invented when ECMA-48 was last updated, so there is no standard support for them in ECMA-48. But some recent implementations based on ECMA-48 do have them via an extension. (Note that hyperlinks can pose a security risk, and measures should be taken to mitigate that risk.)
7. There are no named styles. Just a "style state", in general no stack of them (except for PTX and an extension based on an Xterm extension, as well as math expressions). So, by modern standards, it is quite rudimentary. It is really a low-grade "higher-level" protocol for text styling, except that there is no higher level. The codes are "ignorable" (as well as interpretable) for display of text, at the text level, by design. Even so, it is already quite advanced in some respects, for example it has (for decades) included support, standards-wise, a) for CJK style emphasis and b) CJK "ruby" texts (both supported in modern HTML/CSS) as well as framing of text on a line basis (similar in nature to Egyptian hieroglyph cartouches) also supported in modern HTML/CSS.
8. A type of text formatting which does not have any support in ECMA-48 5th edition is math expressions. One could use **APC** and embed, e.g., TeX math expressions. However, for one thing, that does not cover full Unicode support for math expressions at least not for TeX math expressions. *There is a separate proposal for math expressions.*

In this proposal we also include some additions specifically for enabling easier conversion from ISCII styling and from Teletext styling (including that for Teletext subtitling, which is still a common use for Teletext today) to ECMA-48 styling (these are SGR extensions). The details of such conversions themselves are, however, out of scope for this proposal of additions.

ECMA-48 5th edition actually does not say how to use the control sequences, in particular it does not say how to use styling control sequences. For some other control sequences, there are some indications on how to use them. E.g., cursor movement control sequences are to be sent from the “display component” (originally to be understood as a display with associated keyboard; in a more modern interpretation it would rather be a (display screen) window, this could be a terminal emulator or just the window where a text document is displayed) to the “data component” (to be understood as the software that actually manages a document or a pseudo-document; for instance, text editor programs or text display programs, or programs that has some subcomponents that has text in them, like web browsers with an address line or fields in a form). Note also that there is no requirement that the “display component” itself is handled via ECMA-48 control sequences (like in a terminal emulator), it may well be controlled by other means (like a window text API, as in Windows/XWindow/MacOS/smartphone windows).

For text styling, which is the focus of this proposed update, the styling controls can be used as means of (approximately) display content (with mostly text) which is styled by other means. E.g., programs that display web pages mapping (approximately!) the HTML/CSS styling to ECMA-48 styling. An example of that is the Lynx web browser (http://lynx.browser.org/) displaying (text) web pages, in a very approximate manner, in a terminal emulator.

But the text styling control sequences can also be used as the storage format (in a text document), storing the styling as ECMA-48 control sequences directly. This is then a more light-weight alternative to using HTML/CSS or some other “highly capable” document formats (such as MS Word XML format, ECMA-376-1:2016: *Office Open XML File Formats — Fundamentals and Markup Language Reference*). Often, one does not need all the advanced capabilities of those storage formats for text documents, and a more light-weight format is sufficient. For that, ECMA-48 text styling (as storage format) comes in handy, since completely unstyled “plain text” can be a bit too poor. So ECMA-48 text styling bridges a gap between the “highly capable” text formats, and the “very poor” pure plain text. And that with an existing mechanism. Though old it is still in widespread use, not needing to invent a completely new middle-ground format. Even RTF may be considered too heavy-handed. For terminal emulators, only ECMA-48 text styling is viable. All other currently existing text formatting mechanisms are out of the question for terminal emulators, for technical reasons. In addition, there are also compatibility reasons, ruling out other technically possible (and currently imaginary) styling mechanisms. But here we will focus on storable documents and using ECMA-48 text styling for them in storage, no matter how they are displayed or handled internally in, say, text editing programs. Still, many of the suggested updates here are applicable also to terminal emulators, and certain proposals here are kept in line with what has been done for some terminal emulators.

2 Goals and non-goals for this update proposal

2.1 Goals for the additions and clarifications in this proposal

a) Encouraging existing implementations to support bold/regular/lean font weights in the modern way (no colour change), full colour specifications, use of proper syntax (in particular for full colour control sequences) and other commonly implemented features of ECMA-48 SGR so they are consistent from a user point of view. That is, trying to converge the interpretations of exactly how these should be interpreted (such as fixing commonly implemented colours to be the same RGB values across implementations, just like HTML/CSS named colours are reliable across implementations).

b) Extending the ECMA-48 styling mechanism to other styles or style variants now supported in HTML/CSS, though it will not be as flexible as HTML/CSS, but still in “the same vein” as original ECMA-48. Plus, a minor extension to handle long division notation.

c) Including some commonly supported ECMA-48 extensions, esp. those for colour levels, as well as some extensions for supporting Teletext styling functionality (which is/was commonly used), and some other older styling mechanisms (such as those for ISCII).

d) Moving some styling controls (from other control sequences) to the “**m**” set of controls (i.e., SGR) and generalise them a bit (line and character spacing, font size, font size modification (condensed, extended), and more).

e) Resurrecting HTSA (as HTSA2), “the better way” of setting tab stops. At the same time, other ways of setting tab stops will be deprecated.

f) Specifying PTX better and generalise it, so that table rows with different cell widths (heights if vertical lines) can be used. (Note that PTX, oddly, is also used for “Ruby” text for Japanese/Chinese, though with other parameter numbers.)

g) Extending SPD (as SPD2) (SELECT PRESENTATION DIRECTIONS) for specifying enablement of the Unicode Bidi algorithm (when Bidi processing is provided by the implementation).

h) Arithmetic “by hand” calculation (up to long division) layout is intended to be covered (details will be in a separate document). Doing addition, subtraction, multiplication, and division (between two decimal numerals) is an important part of any modern script. Still, that has not been well covered by Unicode or any other system (though MathML has an attempt at covering long division).

2.2 Non-goals for the additions and clarifications in this proposal

a) To get all implementations of ECMA-48 to necessarily implement all the styling control sequences (indeed, some are not well suited for, e.g., terminal emulators). Likewise for the other control sequences discussed in this paper.

b) Replace other styling mechanisms (RTF, ‘markdown’, …), though support for ECMA-48 styling may be an *additional* mechanism, in particular in text editors.

c) Add “structural” mark-up to ECMA-48 (auto-numbered lists, automatic heading numbering and styling, named styles, etc.) common in more advanced higher-level mark-up mechanisms. Note though that there is already some structural mark-up in ECMA-48, for instance PTX has nesting structure allowing to have tables in table cells.

d) Handle math expression layout and styling. There is a separate proposal for handling math expressions, 1) in an ECMA-48 context (or just text without any styling), 2) HTML/XML context, and 3) other contexts where a “markdown” way of representing math expressions is more appropriate.

e) Handle keyboard, mouse/similar input (and occasional output) and other functionality that is not text formatting related is not covered by this proposal. Those parts of ECMA-48, and extensions in those areas, may be covered in separate proposal.

3 Regular expressions for parsing escape sequences, control sequences and control strings

Originally, the syntax for escape sequences (not to be confused with character references in some programming languages, like \n, \t) and control sequences were formulated as byte 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 regardless of byte encoding.

The general (almost catch-all, we will *here* skip Cd and Cn which are inappropriate for text files) 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 escapes as well as device controls (except **DCS**):

```
c0-char-nondevctrl   ::= [\u0001-\u0003\u0009-\u000D\u001A-\u001F] // skipping Cn and Cd
c0-devctrl           ::= [\u0004-\u0008\u0010-\u0015\u0017-\u0019]  // Cd 'characters'
ESC ::= \u001B  // note: terminal oriented & code page switching escape sequences are all excluded
c1-char-reference ::=  ESC [\u0040-\u005F]
c1-character       ::=       [\u0080-\u009F] | c1-char-reference
cf-character ::=    // note: including line separators, replacement chars and non-characters:
          <characters with general category Cf, Zl, Zp> | [\uFDD0-\uFDEF\uFFFC\uFFFD\UnFFFE\UnFFFF]
SCI ::= (ESC \u005A|\u009A)    // consider all Unicode surrogates, non-characters, and Cf excluded:
sci-sequence ::=   SCI  ( [\u0001-\u0003\u0009-\u000D\u0016\u001C-\u007E] |
         // exclude [\u0016\u001C-\u001F] if the sci-sequence is inside a control string or a math expr.
                             [\u00A0-\U10FFFF]  |  CSI-char-reference )  // +should skip 'surrogates'
CSI ::= (ESC \u005B|\u009B) // using [\u003C-\u003F] first or [\u0070-\u007E] last is private use:
control-sequence ::=  CSI    [\u0030-\u003F]*      [\u0020-\u002F]* [\u0040-\u007E]
```

Non-characters are *object replacement control characters*, each may (per 'paragraph' only) refer to a table (consisting of table rows), an image, a vector graphic or a math expression. Note that a paragraph may contain nested paragraphs (in tables or other objects). *Object replacement control characters* are for runtime internal representation only, not intended for text storage or text interchange.

Character encoding switching escape sequences,
ESC[\u0020-\u002F]+[\u0030-\u004C\u0051-\u0052\u0056-\u007E],
are *excluded* since character encoding switching is outdated and furthermore not allowed to be used with ISO/IEC 10646/Unicode. Any such sequence must be regarded as an error, and be replaced (in conversion or display processing) by SUB or other, visible in display, error indication.

Character encoding switching controls, **LS1** (U+000E), **LS0** (U+000F), **SS2** (U+008E), **SS3** (U+008F), as well as code switching escape sequences, **ESC**[\u0030-\u003F\u0060-\u007E], are *excluded* (and shall be handled as SUBSTITUTE if they occur) since they are not suitable for storing in text documents. Likewise, character references that refer to code switching **SS2** and **SS3** are *excluded* since they are unsuitable for storing in text document, as are all 14 C0 device controls (Cd) and undefined codes (Cn) in C0/U+007F/C1.

Device control strings, starting with **DCS**, are not suitable for storing in a text document, as are **PM** control strings. In addition, cursor movement, terminal screen editing and keyboard control sequences should also be excluded for the same reason, but for brevity that is not expressed in the regular expressions above; but that would entail excluding certain control sequence terminating letters. Cursor movement, terminal screen editing and keyboard control sequences are useful in *other contexts* (like terminal emulators) involving text, but not for storing them in a text document.

A control character, escape sequence, SCI sequence, or control sequence shall not be a code page switching operation, and we have excluded existing ones in the regular expressions above. That is for all contexts, not just text storage. Hence the deletion of intermediary characters, [\u0020-\u002F]*, from the syntax for escape sequences, which was reserved for ECMA-35 (*Character Code Structure and Extension Techniques*) and ECMA-43 (*8-Bit Coded Character Set Structure and Rules*) code page switching. Some C0 and C1 control characters were for code page switching (ECMA-43), but those shall not be interpreted, and have been excluded from the syntax above. Likewise, **LS0**/**SI** (\u000E), **LS1**/**SO** (\u000F), **SS2** (\u008E), **SS3** (\u008F), **LS1R** (**ESC** \u007E), **LS2** (**ESC** \u006E), **LS2R** (**ESC** \u007D), **LS3** (**ESC** \u006F), **LS3R** (**ESC** \u007C), or any other code-switching control, *shall* be uninterpreted and should be treated as SUBSTITUTE. Escape sequences that do not stand for any codepage switching, like control character references, can still be meaningfully interpreted.

There are some control codes purely for certain 1960-ies type terminals (like U+0093, U+0095; but also SYN and START/END OF GUARDED/SELECTED AREA, but these we will generalise so that they are usable in a modern context); those are unlikely to be interpreted anywhere at all today (in particular, Xterm and its derivatives ignore them), and those controls are not recommended for any use, not even terminal emulators. Some control sequences are for keyboard input, and yet other control sequences are for updating a terminal screen/window. The latter two types of control sequences are of course unsuitable to be used in a text document, and should be ignored and handled as SUBSTITUTE if they occur in a text document, especially in an environment where they might otherwise be interpreted (such as a text editor via a terminal emulator); such a text editor may use such control sequences to update the terminal screen/window, but then they come from the editor program, not from the text document itself. The text styling control sequences that are the subject of this proposal can of course be used in a text document that is edited or displayed without any involvement of a terminal emulator; just an ordinary text editor in a modern "GUI"/window environment; just using ECMA-48 styling instead of RTF, a markdown system (there are several), HTML, or some newly invented "private use" markup for storing the text in file, or even as internal representation.

Note that the notation here refers to characters (including that \u0000-\u001F and \u0080-\u009F always refer to ECMA-48 C0 and C1, respectively, *as updated in the Appendices below)*, **not** to bit sequences. The character encoding need not be a Unicode (or ISO/IEC 10646) encoding. If it is in a Unicode encoding, it may be either one (UTF-8, UTF-16BE, UTF-16LE, UTF-32BE, UTF-32LE). But it cannot be an encoding depending on code page switching, nor any encoding that does not cover the ASCII characters (excluding, among many others, national variants of ISO/IEC 646). The latter may be handled by ad-hoc mappings (not specified here).

A control string allows for freer content in the parameter part (compared to a control sequence): the parameters need not be numbers. However, no details of possible contents are given in ECMA-48, 5th edition. So all and any content for these is "private use" in the 5th edition.

DCS ::= (**ESC** \u0050|\u0090) // (Xterm used this to "program" function keys) (*deprecating*)
SOS ::= (**ESC** \u0058|\u0098) // suitable for embedded image data and hyperlinks
ST ::= (**ESC** \u005C|\u009C) // Xterm allowed (*deprecated*) **BEL** (\u0007) as alternative

```
OSC ::= (ESC \u005D|\u009D)   // used in Xterm for setting window titles and such
PM : := (ESC \u005E|\u009E)   // (a message targeted for a “status line”) (deprecating here)
APC ::= (ESC \u005F|\u009F)   // suitable for vector graphics (like HP-GL or SVG)
control-string ::=      // consider surrogates excluded, though not explicit here
  (DCS|SOS|OSC|PM|APC) ([\u0001-\u0003\u0009-\u000D\u001A\u0020-\u007E] | sci-sequence |
     ESC [\u0040-\u0045\u0049\u004B-\u004D\u0054\u0059]  |  control-sequence  |
        [\u0080-\u0085\u0089\u008B-\u008D\u0094\u0099]  |  [\u00A0-\U10FFFF])* ST
```

Also here, for control strings, consider Unicode surrogates, non-characters, and Cf characters excluded from \u00A0-\U10FFFD, which we for brevity express informally instead of in the regular expression. One should only use assigned character positions, but that changes over time, as more characters are encoded.

The contents of control strings are here generalised compared to ECMA-48 5th edition. But **BEL** (U+0007) and **NULL** (U+0000) are excluded. Noteworthy is that **ETX** is the *default* (can be changed) “label” terminator in HP-GL/2, and that URLs (if using a control string for those) may contain non-ASCII characters. Furthermore, control strings cannot nest, and that is expressed in the syntax above. “Labels” in HP-GL/2 correspond to **<text>** elements in SVG. Both allow text to be given in a “plotted”/“vector graphic” component. Styled “labels” should be allowed if HP-GL/2 is integrated via ECMA-48 control strings, and one should also allow math expressions constructed using C0/C1 control codes (see *A true plain text format for math expressions (and its XML compatible equivalent format*), 2025) to be part of “labels”. Since **ETX** (the default label terminator in HPGL) may be used in math expressions, another character, like **SOH** (U+0001), should be used as label terminator.

A prefix of a *potential* match that in the end does not match, should be treated as a SUBSTITUTE. A true match *that is interpreted* is normally not shown as is (except in a show-special mode) but may affect the display of characters that follow. A match (even if not interpreted) must never be case-mapped and should be ignored in collation operations, except for character references (see below).

The following control codes start a control string:

- DEVICE CONTROL STRING (**DCS**, \u0090, **ESC P**), *defunct*, was used by xterm to program function keys (to emit a given string),
- APPLICATION PROGRAM COMMAND (**APC**, \u009F, **ESC _**), possible use for vector graphics (e.g. HP/GL (possibly including C0/C1 math expressions in the ‘label’ strings), SVG, …; inline or via link), font setting (using font name; populate font palette), …,
- PRIVACY MESSAGE (**PM**, \u009E, **ESC ^**), *defunct*, text for a status line in the terminal,
- OPERATING SYSTEM COMMAND (**OSC**, \u009D, **ESC]**), in xterm used for window titles and other data “outside” (not displayed, or have effect, inline) of the document itself,
- START OF STRING (**SOS**, \u0098, **ESC X**), possible use for language tags, author, date of (latest) editing, version number, hyperlinks, or images (inline or via link), …

Each control string is terminated by a STRING TERMINATOR (**ST**, \u009C, **ESC \**) control character.

All of these allow for (almost) any characters between the start and end control characters. ECMA-48 5th edition has for **SOS**…**ST** a bit different specification for the control string part than for the others (**DCS**/**PM**/**OSC**/**APC**). Making that distinction does not seem helpful, so we ignored it here, and all allow the same generalised “content” (in the catch-all-type of syntax given above). Of course, particular instances of control string types, like vector graphics commands, image representation, etc., will impose additional syntax rules. Note again that *this* update proposal does *not* contain any proposals for control string functionality or syntax; just some vague hints (here and below) apart from the *catch-all* syntax (as in ECMA-48 5th edition) above.

3.1 Unicode character references

Escape sequences (**ESC** [\u0040-\u005F]) refer to ECMA-48 (as updated in the appendices here) C1 control characters. They are useful in case the normal C1 controls are not available, or their availability is uncertain. This is a limited form of character references, only for C1 "control" characters. These character references are legacy ECMA-48. The **ESC** character references for C1 characters always refer to C1 as defined in ECMA-48 (as updated in the Appendices below!) regardless of character encoding. This is important especially for character sets that do not have any C1 characters directly.

Also characters with (Unicode) scalar values greater than 009F get character references defined here. We here introduce Unicode character references, in ECMA-48 style, for (Unicode) scalar values greater than 009F. This mechanism is similar to HTML's *decimal* numeric character references. An ECMA-48 decimal character reference is a control sequence:

- **CSI** *n***_**, where *n* is a Unicode codepoint (scalar value) in *decimal* form (hexadecimal cannot be used due to ECMA-48 syntax reasons), *without* leading zeroes and only referring to valid Unicode scalar values larger than 159 (decimal).
  CSI-char-reference ::= **CSI** *<decimal value 160 -- 1114111>* **_** // should skip 'surrogates'…

However, direct character use and using a Unicode character encoding is preferred.

\u009B (**CSI**) is in the C1 space, and is therefore avoided by many terminal emulator implementations, due to the risk of misinterpreting something that is *not* a control code as this control code. Since terminal emulators easily may run in the "wrong" character encoding (compared to what is actually output to the terminal), this is a particularly sensitive issue for terminal emulators. Therefore, a \x9B (ISO/IEC 8859 series) or \xC2\x9B (UTF-8) is often not interpreted as a \u009B, since the encoding setting is not trustworthy in a terminal emulator where an outputting program may blindly output in a different encoding than the one set for the terminal emulator, maybe one that does not even have a C1 area; something to consider also in encoding conversion, using **ESC** character references when converting C1 controls to an encoding that does not have a C1 area. Though far from a complete solution, one is then only interpreting the **ESC [** alternative for **CSI** in many terminal emulators (and similarly for other C1 characters). This uncertainty in encoding for terminal emulators is also the reason why all (useable) device control characters (14 of them) are in the C0 area, none in C1, and **ESC** character references are not ok for device controls.

The situation is different for text files with formatting/styling in the ECMA-48 way, where using C1 characters directly should not be a problem if the file is coded in UTF-8 or UTF-16BE/UTF-16LE. However, if one is using an older character encoding for the file, one may have to use the **ESC** variants for ECMA-48 C1 characters, as many old encodings do not have the ECMA-48 C1 characters.

3.2 General additions for control sequences

Here we reserve **=** (EQUAL SIGN), postfix, for denoting a negative value of a parameter. We cannot use HYPHEN-MINUS as that is reserved as an "intermediary character" that can occur only before the final character of the control sequence. We cannot have **=** as prefix, since then it then may come right after the CSI, and then **=** is reserved for private use control sequences. And we reserve **?** (QUESTION MARK) for denoting decimal separator, with at least one digit before the **?**. Note that also FULL STOP and COMMA are "intermediary characters" associated with the final character. When <, =, >, or ? do *not* occur *first* after the **CSI**, they are *not* reserved for private use, but when one of those four characters occur *first* after the **CSI**, the control sequence is reserved for private use (a few such private use control sequences are defined for Xterm).

3.3 Control strings: Some *possible* uses of APC and SOS (*not* part of proposed update)

Some *possible* uses (*not* part of the proposal here) for APC and SOS control strings are:

- *Declare character encoding*. We here suggest (cmp. HTML) **SOS charset=**<*IANA charset-name*>**ST**, default: **UTF-8**. This declaration should be at the beginning of a document. In contrast to ISO 2022 escape sequences to *change* character encoding, this declaration does not change to any encoding, just *declaring* it (like for HTML).
- *Declare language*. We here suggest **SOS lang=**<*IANA language tag*>**ST**. No default value.
- Embedding *images* (direct or via hyperlinks), like **SOS** [**w=**<*width*>**;h=**<*height*>**;**]**img:**<*link to image*>**ST**; the (optional) width and height are values with unit (incl. *em*, and percent).
- Embedding graphics as *plotter commands* or *vector graphics.* Here we suggest these possibilities:
  1) Use HP-GL/2: **APC** [**w=**<*width*>**; h=**<*height*>**;**] **hpgl=**<*HP-GL/2 commands or* `.hpgl` *filename*>**ST**. HP-GL can in principle, when used in an otherwise ECMA-48 context, be extended to use ECMA-48 styling for "label" (text) strings. Likewise can **SCI**-based math expressions (see *A true plain text format for math expressions (and its XML compatible equivalent format*) be permitted in "labels", but then use **SOH** as "label" terminator.
  2) Or, despite the apparent anachronism, use SVG with inline CSS: **APC** [**w=**<*width*>**; h=**<*height*>**;**] **svg=**<*SVG markup or* `.svg` *filename*>**ST**.
  3) Some other *non-proprietary* storage format used by programs listed in https://en.wikipedia.org/wiki/Comparison_of_vector_graphics_editors *may* be possible to support as vector graphics formats used between **APC**…**ST**.
  Many programs output "ASCII art", assuming a fixed width font is used, to display certain graphics on terminals or resort to produce the graphics (e.g. as SVG) to a file to be displayed by *another* program. By allowing the use of this kind of embedded (and "vector") graphics, the graphics can be output (also) directly to a terminal, as well as embed the graphics in a *.txtf* file.
- Define (possibly parametrised) macros via **APC**, that then can be used via control strings referencing them (and possibly giving parameters). Such macros could then be "called" by follow-on **APC** control strings. No details are worked out here, since this is only listed as a possibility, not a part of this proposal.
- There is an existing hyperlink proposal (implemented in some systems) that use **OSC** instead of **SOS**, though the latter would have been a better fit. That proposal is *not* echoed below, since *this* proposal does *not* propose *any* particular control string.

3.4 Escape sequences and control sequences not suitable for in-text use

ISO/IEC 2022 escape sequence which were used for code switching, are disallowed. Such escape sequences shall be ignored (but displayed like SUBSTITUTE at least for the ESC character). Other codepage switching controls shall also be ignored, and all are unsuitable to occur in any text. Also escape sequences for reset or lock/unlock keyboard are disallowed (but are still useful for terminal emulators, not for text). Certain control sequences in ECMA-48 are edit position movements, screen redraw commands, interrogations/replies to/from a device (terminal) from/to an application, or other uses that is not text formatting. ***Such*** control sequences *should never* be part of any "plan text formatted" document and shall be ignored (but displayed like SUBSTITUTE at least for the CSI part) if any occur in a text document.

4 Ink-less (mostly) characters

The handling of line breaks and spaces, and a few other characters is important enough for the rendering of text that some guidelines are called for. Many of the C0 and C1 "control" characters are not given correct properties in Unicode, especially for bidi, line breaking, and spacing. We will skip over device controls (general category Cd, see Appendices B and C). Uninterpreted (including undefined) "control" codes and control sequences and control strings shall **not be invisible** but should ***display*** as SUB (or REPLACEMENT CHARACTER) or some other clearly visible representation (like the CSI as SUB and the rest of the control sequence as-is, which is consistent with how ill-formed control sequences should be displayed).

Note that the characters below contrast to edit position movement commands, e.g., CURSOR NEXT LINE which is a cursor movement command (for text displaying, and perhaps text editing, applications). Control sequences for edit position movement (or extension/shrinking of text selection) are not suitable to be part of text. There is unfortunately no quick and easy distinction in ECMA-48 whether a control sequence is fit for "in-text" use or not, other than enumeration. The characters below, however, if entered from a keyboard (or entered by pasting) insert these characters in the text, if working with a text editing application or some kind of field that allows text entry. However, some line break characters may be interpreted as an "execute" control in applications such as command line interpreters.

4.1 Space-ish characters: HT, HTJ, SP, NBSP, NNBSP, ENSP, IDSP, …

The following characters "move" the display position (and for HTJ also moves the display position of some preceding characters) in the character progression direction and there should be no glyph lookup (except possibly for OGHAM SPACE) via the font (unless in a "show special" mode, where there is also a glyph for these, but that may be taken from another font and not related to the character via the font, just via the application).

- U+0009; CHARACTER TABULATION (**HT**). HT moves *at least* 1 en, to the next tab stop after that in the character progression direction. HT does *not* stretch when justifying.
  This was originally intended for "rapid form fill-out", where the paper had the actual form pre-printed, and just filling in the field data on the printer, just like for mechanical typewriters. But it soon gained a different use: rapid spacing for non-forms.
  Note that **CSI** [*n*]**I** (CURSOR FORWARD TABULATION, CHT) and **CSI** [*n*]**Z** (CURSOR BACKWARD TABULATION, CBT) are *cursor movement* commands when editing a document, if keyboard input is given as ECMA-48 control sequences; they do not insert tab characters into the document, nor should these control sequences occur in a document.
- U+0089; CHARACTER TABULATION WITH JUSTIFICATION (**HTJ**, as a character reference: **ESC I**). HTJ not only moves to the 'next' tab stop in the character progression direction (but moves *at least* 1 en) like HT, it also moves some *preceding* characters (back to the nearest of: beginning of text, nearest preceding (NLF or automatic) line break or nearest preceding HT/HTJ *character* (not preceding tab stop as originally defined) in the same

displayed line) to align the end of that text with the next tab stop at least 1 en after the current position (movement is in the current character progression direction).

HTJ is useful for alignment of numeric literals (with the same number of decimals) on several lines or aligning item labels in a text items list. HTJ must act as HT for line breaking and bidi. Correct line break property is BK, correct bidi property is S.

- U+0020; SPACE (**SP**). The width of SP, using a proportional font, is often less than en width, and can stretch for automatically justified lines.
- U+3000; IDEOGRAPHIC SPACE (**IDSP**). Nominally 1 em wide but may stretch in automatically justified lines.
- U+1680; OGHAM SPACE MARK. May have a glyph; if glyph is missing it acts as an SP. (No glyph for an OGHAM SPACE MARK that is in a sequence of spaces that is followed by a line break (hard or automatic).)
- U+00A0; NO-BREAK SPACE (**NBSP**). Same width as SP but does not allow for automatic line break. NBSP can stretch for automatically justified lines just like SP.
- U+2007; FIGURE SPACE (no-break). Should be the width of a “0”, esp. if fixed-width (“tabular”) digits.
- U+2008; PUNCTUATION SPACE (**PSP**). Should be the width of a FULL STOP or COMMA glyph (current font and font size), unfortunately it is not no-break; see NNBSP.
- U+202F; NARROW NO-BREAK SPACE (**NNBSP**). Should be **PSP** (U+2008) width. Suitable as space between digit groups (usually of 3 digits).
- U+0082; [LINE] BREAK PERMITTED HERE (**BPH**, as character reference: **ESC B**). Handle equivalently to U+200B ZERO WIDTH SPACE. Makes explicit a position for allowed automatic line break. Correct line break property: ZW. Compare the control **SHY**.
- U+0083; NO [LINE] BREAK HERE (**NBH**, also as character reference: **ESC C**). Handle equivalently to U+2060 WORD JOINER. Correct line break property: WJ.
- U+200B; ZERO WIDTH SPACE (**ZWSP**). Handle equivalently to **BPH**.
- U+2060; WORD JOINER (**WJ**). Zero-width (if supported) no-break word *separator*, despite the name. Handle the same as NBH (if both supported).
- U+FEFF; ZERO WIDTH NO-BREAK SPACE (**ZWNBSP**). The original replacement for NBH, but was co-opted as “byte order mark” (iff very first in a text *file*; not if first in a string in other kinds of string storage, and when concatenating files, all but the resulting first ZWNBSP will lose their “byte order mark” significance). This character is not recommended to use at all, especially not as a “byte order mark”; render as SUB.
- U+00AD; SOFT HYPHEN (**SHY**); character reference: **CSI 170_**. A zero-width space but explicitly indicates a possible hyphenation position. If a hyphenation is done at that point, SOFT HYPHEN displays as <HYPHEN, LS> in most scripts, maybe more characters depending on the hyphenation rules.
- U+2000; EN QUAD (canonically equivalent to U+2002 EN SPACE).
- U+2001; EM QUAD (canonically equivalent to U+2003 EM SPACE).
- U+2002; EN SPACE (1/2 em).
- U+2003; EM SPACE. Nominally approximately the width of an “m”. But better, it is the height of “Åg” or similar letters (and some fonts may need size adjustment).
- U+2004; THREE-PER-EM SPACE (1/3 em).
- U+2005; FOUR-PER-EM SPACE (1/4 em).
- U+2006; SIX-PER-EM SPACE (1/6 em).
- U+2009; THIN SPACE (1/8 em).
- U+200A; HAIR SPACE (1/24 em).

- U+205F; MEDIUM MATHEMATICAL SPACE (4/18 em, 1/4.5 em).
- U+180E; MONGOLIAN VOWEL SEPARATOR (zero-width when supported and between Mongolian letters, but may change the shaping of adjacent Mongolian vowels).

4.2 Line break characters: LF, CR, CR+LF, PS, LS, VT, FF, IS*n*, STX, ETX, …

The following characters "just" advance the display position in the line progression direction plus to a beginning of line position in the character progression direction. Just as for most "default ignorable" Unicode characters these should have no glyph lookup via the font (unless in a "show special" mode, where these characters also have glyphs, but may be taken from another font and not related to the character via the font, just via the application).

- U+000A; LINE FEED (**LF**), originally pure line feed (and still so in terminal emulators), now handled as equivalent to **CRLF**/**NEL**/**PS** in text files (and SMS). (With default *stty* settings in Unix/Linux, an output LF is converted to CRLF.)
- U+000D; CARRIAGE RETURN (**CR**) (entered by *return* key) often mapped to LF/CRLF or (rarely) even **NEL**; but sometimes still interpreted as a pure carriage return, e.g., in "raw mode" *tty* output (Unix/Linux). The line break property *CR* should be interpreted: "Cause a line break (after), except between **CR** and **LF**, **CR** and **VT**, **CR** and **FF**". Note that CR is a *filler* character (there is no NULL) in SMS charsets, both 7-bit and UTF-16 SMS encodings.
- <U+000D, U+000A>; CARRIAGE RETURN, LINE FEED (**CRLF**), must be handled as a single line break; it is *not* two (explicit/hard) line breaks, it is just one. Note that <LF, CR> is not handled as a single unit, but is two "hard" line breaks.
- U+000B; LINE TABULATION (**VT**) (often possible to enter from a keyboard as *shift-return*), **VT**/**CRVT** (the latter as a single unit) must act as LS for bidi.
  VT was originally intended for rapid form fill-out (where the paper had the actual form pre-printed, and just filling in the field data). But it has gained a new use: to act as LS, in C0.
- U+000C; FORM FEED (**FF**) (sometimes possible to enter as *ctrl-return*), **FF**/**CRFF** (the latter as a single unit) must act as LS for bidi. Correct bidi property: S (just as for **VT**). Inside a table cell, math expression, vector graphics or there otherwise is no pagination, FF has the same effect as VT.
- U+0085; NEXT LINE (also as character reference: **ESC E**) (NEL), was intended as replacement for **CRLF** but has been actively used only in EBCDIC based environments. Must act as **PS** for bidi.
- U+2029; PARAGRAPH SEPARATOR (**PS)** Was intended to disambiguate **CR**/**LF**/**CRLF**.
- U+2028; LINE SEPARATOR (**LS**) Was intended to disambiguate **CR**/**LF**/**CRLF**. But now, instead, **VT** is often used in this sense (LS) rather than VT's original sense.
- U+001C, U+001D, U+001E, U+001F; **IS4**, **IS3**, **IS2**, **IS1** (these should *normally* not occur in text). Each of these should display as a line break, perhaps with an extra mark (e.g., encircled "IS4", …, "IS1", or set preference). The use of **IS1** to **IS4** is *application defined*, they are private-use data separators. Corrected properties for all four: line break: BK, bidi: B.
- U+0001, U+0002, U+0003; **SOH**, **STX**, **ETX**, these were apparently intended for a telegram-like protocol. Each of these should display as a line break, perhaps with an extra mark (e.g., encircled "SOH", …). Corrected properties for all three: general category Zp, line break: BK, bidi: B. These three should reset SGR to the inherited values. **STX** and **ETX** will be reused for delimiting (short) text parts *within* math expressions.
- U+008D; REVERSE LINE FEED, like BACKSPACE (and to be nitpicking, also CR) this was probably intended for overtyping or math expression buildup (neqn/nroff).
- U+0091, U+0092: PRIVATE USE ONE, PRIVATE USE TWO; regard these as INFORMATION SEPARATOR FIVE and INFORMATION SEPARATOR SIX; still private use, but reasonably limited to be 'information separators'. Recall that IS1…IS4 are private use information separators, they are *not* necessarily the ASCII UNIT SEPARATOR to FILE SEPARATOR.

- U+0086, U+0087, U+0096, U+0097; in 5th edition of ECMA-48, these have a semantics very much tied to so-called block terminals. We generalize them to INFORMATION brackets, letting the block terminals semantics be a specific instance of these brackets, generalising them like the ISs are.

See Appendices B, C, and D for a list of improved Unicode character properties for C0, C1 and "non-characters" characters; the properties that are found in UnicodeData.txt and LineBreak.txt. Se Appendices E and F for more details on device control characters and INFORMATION SEPARATORs/BRACKETs respectively.

Many applications today implement automatic line breaking (and automatically re-evaluating the line breaking whenever needed). Certain control sequences (proposed to be deprecated) assume that all line breaking is done by hard line-breaks (i.e., explicit line break characters). But much of ECMA-48, esp. SGR, work well also when line-breaking (or, rather, line wrapping) is made automatically. Note that when doing automatic line breaking, one should try to avoid local "jaggedness" in line length, if possible, like done by TEX via a points system, as well as by automatic hyphenation (which is language dependent).

There is some ambiguity as to how to represent a "paragraph break" vs. an explicit "line break". The recommendation here is: LF, CR, CRLF, NEL, PS should be regarded as equivalent, and (CR)VT, (CR)FF, LS should be regarded as equivalent (though (CR)FF also still often in addition retains its original interpretation). VT is commonly used in place of LS in many applications.

The interpretation of the style set by **CSI 69:…m** and **CSI 70:…m** (both new) are affected by the difference between PS (etc.) and LS (etc.): namely whether a line is considered as a:

- paragraph start line: text just after beginning of text, LF, CR, CRLF, NEL or PS,
- not paragraph start line: text just after automatic line break, (CR)VT, (CR)FF or LS.

The distinction also affects the Unicode bidi algorithm, when that is enabled (see SPD2 below).

4.3 Replacement characters (these have 'glyphs'!)

- U+001A, U+FFFD; SUBSTITUTE, REPLACEMENT CHARACTER, these two should *display* the same, *with* some glyph. Used for conversion errors and other errors like using a device control in a text file or a malformed or an (in a particular use) uninterpreted control (sequence). Correct properties (for both): line break: AI, bidi: ON, general category: SO.
- U+FFFC; OBJECT REPLACEMENT CHARACTER and all "*non-characters*". These are not spaces but are placeholders (per paragraph!) for (parsed out): math expressions, tables, vector graphics, images and perhaps other kinds of "objects" (e.g., "ligated" emoji sequences). These placeholders should never occur in "interchange" or text file storage and should effectively be replaced by SUBSTITUTE (or REPLACEMENT CHARACTER) if they do. Each paragraph internally has a table saying which non-character stands for which "object" in the paragraph, and of course the "object" (math expression, table, …) itself. The number of "non-characters" in Unicode is unfortunately a bit stingy, even though the "scope" is only one paragraph. An application that really needs more of them can use a small part of the PUA (private use area), e.g., the one on plane 16. Correct line break property (for all): CB, correct bidi property: ON, correct general category: SO.

4.4 SUB is not ink-less

SUBSTITUTE (SUB) is "used in the place of a character that has been found to be invalid or in error. SUB is intended to be introduced by automatic means". While a bit fuzzy, that means that it is intended as an error indication, not some (in display) invisible character. Indeed, Unicode introduced a technically equivalent character REPLACEMENT CHARACTER and made sure to make clear that it is a visible character. SUB (and REPLACEMENT CHARACTER) are intended to be “introduced by automatic means”, like when trying to convert a character to another encoding but the target does not have an equivalent character, or when finding a wrong encoding for UTF-8 or UTF-16 (like an “isolated surrogate”), or when a character is not supported by the current (or fallback) font. W.r.t. ECMA-48, SUB (or REPLACEMENT CHARACTER, if available) should be used for escape sequence (ESC-based), control sequences (CSI or SCI based), and control strings that are malformed, undefined (but well-formed), or unsupported (well-formed, defined (ECMA-48 or an update to it), but not supported by the implementation). While REPLACEMENT CHARACTER often does have glyph in fonts, that is not necessarily the glyph to use for SUB or REPLACEMENT CHARACTER. One can give a “hex box”, which is often seen in Linux applications, and control sequences/strings are not in error or are not interpreted can display the control sequence/string in its entirety. It is still nominally an “automatically introduced SUB” that has a “full display” to indicate that something is amiss.

Control codes/sequences/strings may be (depending on the proper effect) invisible *assuming that* 1) the control code/sequence/string is at all supported, *and* 2) there are no syntax errors (including any syntax that is added for that particular kind of control sequence/string, like for instance SVG syntax for a control string that is supposed to hold an SVG graph, or variant parameters in a styling control sequence) in the control sequence/string. Otherwise it is *visible* as an error indication. Ideally, the error display should be informative, and should be the same as in a “show invisibles” mode which show also controls that are interpreted.

4.5 Misc

- U+0000, was NULL. Nobody uses the *character* NULL, but the value 0 (in almost whatever character encoding; notable exception: SMS/CBS character encodings) is often used for special purposes, like string termination, or to say “not set” (but can be set to a single character). Hence we note U+0000 as reserved, but not a character (NULL).
- U+0016, was SYNCHRONOUS IDLE (**SYN**). SYN was a replacement for **NULL** (as a “time filler” over a communication line), but also **SYN** is now fully outdated.
- BACKSPACE, was U+0008 here proposed moved to U+0094; BACKSPACE basically presumes a fixed-width font, can be used for glyph overstrike, often for underlining or “bold” (assumes adding actual ink…). BACKSPACE is hardly ever implemented anymore. U+0008 (mapped to whatever encoding, including UTF-8) has now the semantics of CANCEL CHARACTER. So we propose to move CANCEL CHARACTER to U+0008, that that is the de facto semantics of U+0008 now. Swapping in BACKSPACE to U+0094 to keep it, since it is not defunct (note though, that BACKSPACE does not erase anything).

5 C/C++ `int iscntrl(int c)` and friends

The description for the standard `int iscntrl(int c)` function in C/C++ currently says:
“A *control character* is a character that does not occupy a printing position on a display (this is the opposite of a *printable character*, checked with isprint).
For the standard ASCII character set (used by the "C" locale), control characters are those between ASCII codes 0x00 (NUL) and 0x1f (US), plus 0x7f (DEL).”

However, this description is **wrong** and based on a troublesome misunderstanding. The "does not occupy a printing position on a display" is an old and very unfortunate misconception. When a control character is indeed *interpreted*, it often does not take up any 'printing' space (in more modern terms: zero-width *and* ink-less). However, HT, LF and more *do* take up space (though nominally inkless, except for a 'show invisibles' display mode), but also those are sometimes not interpreted (like in a one-line field in a GUI interface, like in Linux).

But when not interpreted, a control character *shall neither be inkless nor non-spacing*. Indeed, such a character should be displayed as SUBSTITUTE (which is never "invisible" but is instead functionally equivalent to REPLACEMENT CHARACTER). Another *visible* display is permitted, for instance like the "hexadecimals boxes" which is common in Linux. But *non-interpreted* characters *shall **never*** be invisible in display. Likewise, for *uninterpreted* or *undefined* control sequences or control strings. (Certain characters are invisible *when properly interpreted*, as are *interpreted* control sequences and control strings.)

The description for the standard function `int iscntrl(int c)` in C/C++ (and corresponding in other programming languages) should be corrected; Cd, Cf (incl. SHY), Zp, Zl characters and so-called *non-characters*, as well as SUB and (OBJECT) REPLACEMENT CHARACTER should return 1 (true) for `iscntrl`. Indeed, also a small number of math operators (in addition to MHD: also BIG SOLIDUS and similar characters) should be regarded as control characters (see separate math expression proposal), and return true for `iscntrl`.

`iscntrl` and related methods are currently in general locale dependent in C/C++, but for a *UTF-8 locale* **require** that the argument is a *Unicode scalar value*, **and** that the returned value is **not** locale dependent for UTF-8 locales. For a UTF-8 locale, *do disregard* the locale file content for character properties; instead, it is the Unicode properties that are considered, updated as per Appendix B below. `iscntrl` is *not* simply what is (currently) Cc in Unicode (and in addition we practically eliminate that general category in Appendix B) but also includes Cf characters as well as some Sm and So characters. Ideally, for a non-UTF-8 locale, `iscntrl` should still depend on Unicode character properties (as updated here) and not locale data, though the argument to the functions discussed here would then be in the (single-byte) locale encoding.

For `int isprint(int c)` (and `isgraph`) in C/C++ (and other programming languages). The name is an old misnomer. Control characters are often "printable", in particular uninterpreted/undefined ones, but also LF, HT and more are usually printable (often, but not always, inkless). `isprint` returns true for L*, N*, P*, S*, M*, Z*, CB, HT, HTJ (as updated in Appendix B) including SUB and MHD. `isgraph` is like `isprint` but excludes Z*. `isspace` returns true only for Zs (excluding HT) and `isblank` returns true for Z* (as updated in Appendix B and C) and HT (but not HTJ). Undefined code positions (Cn; w.r.t. the Unicode version supported) return false for all of these, though they are printable (as a *.notdef* glyph, SUB, REPL. CHAR., or "hex square"). Note that that Unicode has several "space" (Zs) characters, at least one of which (OGHAM SPACE) often has ink and all may have ink ('show invisibles', which is a common editor display mode). For *all* of these functions, and a UTF-8 locale, they should return 0 (false) for integer values that are not Unicode scalar values (too large or reserved for UTF-16 or negative).

Going a bit more off-topic, but related: None of the "character type" test functions in standard C and C++ are well adapted to Unicode characters (use a Unicode library) but should still, for a UTF-8 locale, ignore what is in the locale file and instead be based on Unicode categories. `isalpha` returns true 'only' for L*, Sk, Mn, Mc, Nl, No. `isdigit` returns true only for Nd (decimal digits). `isupper` returns true only for Lu and Lt. `islower` returns true only for Ll. `ispunct` returns true only for P*, S* (except Sk). `isxdigit` returns true only for certain ASCII and "FULLWIDTH" characters (as always: uppercase and lowercase A-F, and digits 0-9; no other prestyled digits or letters (excluding in particular "MATHEMATICAL" characters), nor digits or letters in any other scripts than Latin). No UTF-8 locale thus needs to contain any data for these functions and should omit such data, which would be large (to cover all of Unicode, which current locales do not), repetitive and the same for all UTF-8 locales.

There is no standard (C/C++) function for checking for "newline function" (NLF), a concept used in Unicode as well as here. But an appropriate regular expression for (general) NLF is (\r|\r?[\n\v\f]|{nel}|{ls}|{ps}). However, that does not distinguish between PS-like and LS-like NLFs, nor are all BK characters NLFs. These are different both for bidi as well as for layout. LS-like NLF: (\r?[\v\f]|{ls}), PS-like NLF: (\r|\r?\n|{nel}|{ps}). Note though that PS is intended as a paragraph *separator*, while \n is usually regarded as a paragraph (or line…) terminator. For a "running text" document, that distinction is less of an issue, than for, say, a script source file where \n (or \r\n) is often (not always) interpreted as a signal to execute the command on given line of code. In addition, NEL, LS and PS are rarely supported at all (likewise for lone \r), and in some contexts one might either have to use \n rather than \r\n or vice versa. When using multiple platforms, it is more user friendly to allow a variety of NLFs mixed, than to force conversion or 'normalisation' of NLFs, except when backward compatibility reasons force the use of only a particular NLF (often only \n, only \r\n, or rarely only NLF). For this proposal we will assume the general regular expression when writing "NLF", but different contexts may need to narrow that. For (Unix/Linux oriented) terminal emulators, the "original" interpretation of \r and \n holds, and to get proper newline from \n one need to use "cooked mode" for the tty, actually inserting a \r just before each *output* \n; but for input the tty *instead* converts \r to \n; on Windows, instead an input (from the keyboard) \r (CR, even if by the corresponding keycode) is instead turned into \r\n (CRLF, at least when targeting a text file being edited; some text editor programs have a user preference for which newline mapping, or conversion, is done).

There are several other control codes (SOH, …, IS1, …) that should display as line break. However, these we will consider to be akin to cell boundaries in a table, thus considered as *component boundaries*, like actual table cell boundaries (see control sequences below), and not as NLFs. Note that any NLF would be *inside* a cell as defined by component boundaries. Styling changes in within a cell does not cross over a cell boundary.

6 Unicode and ISO/IEC 10646 mistreatment of C0/C1

Both Unicode and ISO/IEC 10646 treat the C0/C1 areas with very much disdain. Unicode regards C0/C1 areas as basically "private use areas". ISO/IEC 10646 instead refers to the defunct registry of "control codes"; this registry is *not maintained*, is used by *absolutely nobody*, and presupposes the use of escape sequences to switch between "registered sets"; while explicitly referenced by ISO/IEC 10646, these escape sequences are implemented *absolutely nowhere*; they are also absolutely opposite to the design of Unicode and (the rest of) ISO/IEC 10646. The registered "sets" themselves are a mess: one is for ECMA-48 (but with a typo), many of the others are just small subsets of ECMA-48 controls (pointless), one is supposedly for Teletext but is completely wrong (there is a formal standard for Teletext), and the remaining are a progression of (i.e. updates to) collation indicators for collation of library records (should be done by other means, such as using ECMA-48 **IS***n*, **PU***n*, private use control sequences, or (nowadays) Unicode PUA characters, or more likely CSV or JSON designed for the purpose.

The laxer (but code switching escape sequence free; at least formally) Unicode approach of laissez-faire regarding C0/C1 for such things as EBCDIC, ATASCII (ATARI), PETSCII, and many other old (and now completely unused) encodings are just an unhelpful mess. Defining mappings to standard or private use ECMA-48 control sequences, or Unicode PUA characters is a much more viable approach, for those who need to handle such character encodings in a modern setting (maybe using Unicode). (This implies that the ICU (Internationalisation Components for Unicode) mappings for EBCDIC should be revised; other old character encodings are not covered by ICU.)

We strongly suggest that C0/C1 characters should be regarded as standardised and fixed also for Unicode and ISO/IEC 10646, with the updates as per appendices. In practice "everyone" (almost) does that already; anything else would spell the death of all modern computing. However, before fully settling them, we do need to fix (correct) some of the Unicode properties to get that to work well, and also reserve some unused control codes for math expressions. In particular: general category (the current general category Cc is extremely unhelpful and will therefore be eliminated), bidi properties (there is a partial attempt in Unicode to handle that for C0 characters, but falls short), and line break property (again, there is a partial attempt in Unicode to handle that for C0/C1 characters but also falls short). We will focus on these Unicode properties (others may need corrections as well, for instance that the name for U+0007 cannot be BELL, some aliases are misleading, and more), and we list proposed actually useful property values for these three properties and C0/C1 characters in Appendix B, and more formally for *UnicodeData.txt* in Appendix C and for *LineBreak.txt* in Appendix D.

In the math expression format proposal (C0/C1 version) there is also property updates for some math control characters (such as BIG SOLIDUS) suitable at least for inside of C0/C1 math expressions. (A few "operator" characters are treated as control characters, since they have layout effects.)

7 Intermission… and deprecation of *all* ECMA-48 "modes"

The sections above give general clarifications and additions to the basics of ECMA-48 control codes. We will in the following sections focus on updating the part of ECMA-48 that deals with styling of text. Updates to other aspects of ECMA-48 is not part of *this* proposed update.

ECMA-48 "mode" changes **shall** *not* be supported. GRCM – GRAPHIC RENDITION COMBINATION MODE must be fixed to CUMULATIVE. TSM – TABULATION STOP MODE must be fixed to MULTIPLE. Further, GCC – GRAPHIC CHARACTER COMBINATION (a kind of character combination; **CSI** … **SP _**) **shall** not be implemented, nor should BS or CR be used for overtyping, except for typewriter-like devices (which are very rare today) and underline/bold; Unicode has other ways of composing characters, and here we give other ways of underlining, overstriking and make bold. RM – RESET MODE (**CSI** …**l**; small ell) and SM – SET MODE (**CSI** …**h**) also **shall** not be implemented.

8 SELECT GRAPHIC RENDITION – SGR (extended and clarified)

SGR controls steer how to render glyphs, from a font, for the characters targeted (the control sequences are inline, and the characters targeted are implied). Some characters, in particular controls and spaces, do *not* look up a glyph for that code point in the given font (most spaces just compute a width) or use a glyph for another character (for SUB or 'show invisibles'). For characters for which a glyph is to be looked up, but the font does not have one, nominally a SUB glyph is displayed, but a clearer display can be used, like the hex boxes used in some Linux GUIs. (The *.notdef* glyph of OpenType fonts, is a poor display of SUB.)

Emoji are not looked up in a regular font, and not sensitive to some of the styling controls, e.g. insensitive to text colour setting, except transparency and colour negative. And there are other styling restrictions, like that italics/oblique do not affect symbols (S*, but do have an effect on punctuation, P*), certain styling does not affect prestyled characters, e.g., MATHEMATICAL BOLD CAPITAL A is bold and upright regardless of weight and italics settings. Prestyled as superscript characters are displayed as first level superscripts, regardless of superscript/ subscript setting.

A single logically contiguously styled substring may be split into multiple substrings of that style if bidi is enabled, due to bidi reordering. Note that bidi processing can be turned off, fully or partially (see below), dynamically or by preference setting. Limiting bidi is essential for source code (like CVS, JSON, XML, but also for any programming or scripting language) viewing or editing when there are bidi text parts in the data/code, and is an absolute requirement for math expressions. "Full" bidi is intended for plain prose, not for data formats nor programming languages. Styles that are set *within* a table cell, *within* a math expression text part, or *within* a vector graphics (SVG, HPGL, PlantUML, …) text part are reset at end of such a text component, and such components are also bidi isolated (as if the delimiting control sequences had bidi category B).

The SGR control sequence can specify several rendition changes in a single control sequence. However, **CSI** *a*;*b*;*c*;…;*x*;*y*;*z***m** can be converted to **CSI** *a***m CSI** *b***m CSI** *c***m**…**CSI** *x***m CSI** *y***m CSI** *z***m** (with no semicolon within the variables, but here they may contain colon, the space between **m** and **CSI** is part of the notation here, not to occur literally). This kins of split up is valid for SGR but not in general for control sequences.

Note that when (and if) bidi processing is applied, a single original "span" with a particular "graphic rendition" (called "style" today) may be split up into *several* display spans with *other* display spans having other styles *in-between*. How to manage and represent that in an implementation is out of scope for this proposal but needs to be handled by an implementation of ECMA-48 styling that also implements bidi processing.

Here we go through the SGR codes, existing in ECMA-48 5th ed. as well as here proposed extensions, with additions and clarifications, grouped by function, not just listed numerically.

8.1 RESET RENDITION (extended, primarily for command line apps)

This control sequence resets the SGR settings to the default (which may be preset or set in preferences). What is the default is out of scope for this proposal (as it is for ECMA-48 5th ed.). If pushing SGR attributes is supported (see below), the top-of-stack of stored-away values are counted as default. Reset rendition should be avoided in text documents and contiguous text output. This is a shortcut that can be handy, especially in the context of command line interpreters, not needing to reset the style changes one by one, especially between commands.

This is useful in cases where the "current SGR state" is not or cannot be known by the application, and a reset is desired. For instance, at the *beginning* of a terminal program's prompt (after a very first **ST**, just in case the previous output was terminated in the middle of a control string).

- **CSI 0m** Reset rendition. The 0 can be omitted (**CSI m**). Resets rendition attributes (set via SGR) to the default setting and may (should) also close nesting layout features like PTX (tables/Ruby), bidi controls (ECMA-48 or Unicode) and other nesting features (like for math expressions). It should not close explicit stacking of SGR settings (à la *xterm*, see below). The exact extent of what is reset is implementation defined. The main use for **CSI 0m** is for terminal emulators and programs that run via terminal emulators, though it can be handy (but not necessary) for dealing with conversion from ISCII (to the extent its formatting was ever implemented) or Teletext (which commonly uses colouring); except for these two, **CSI 0m** should not be used in (ECMA-48 formatted) text files.

Hard line breaks do not, according to the ECMA-48 standard, reset the rendition set by SGR control sequences. However, some spacing setting control sequences, here proposed *not* to be used (replacements in SGR are proposed below), are auto-reset on (hard) line-breaks.

A prompt (for a command line interpreter) should *begin* with **ST** (to close any control string that may happen to be open) and then **CSI 0m**, followed by the actual prompt. **CSI 0m** should not be used in any other context (with the exception of ISCII or Teletext formatting being converted, where using **CSI 0m** is handy, but *not* necessary). **CSI 0m** is the only control sequence in this proposal that is specifically for terminals (or, rather, command line interpreters). All the rest in this proposal is suitable for a text document format, and the UI styling controls suitable for a text document editor. However, ISCII formatting and Teletext formatting do formatting reset when moving to next line (or page; so when emulating those, e.g., due to character coding conversion, a **CSI 0m** is suitable just after an (explicit) line break (or explicit page break), to make the emulation correct.

A note for terminal emulators (command line interpreters): Some programs (esp. those outputting partial content of large text files or text streams) currently do assume, or even do, effectively reset the SGR set rendition, inserting extra **CSI 0m**, and more, on (some) hard line breaks. That may result in somewhat “style thwarted” display of some portion of text, compared to if the entire preceding part of the document had been read and been ECMA-48 interpreted before display, if the document has styling that spans more than (say) a few lines or “paragraphs”.

8.2 FONT WEIGHT (clarified, extended with variants)

Font weight is the scale where the boldness or leanness of a font is given, there is even degrees of boldness or leanness. Most modern fonts have a normal weight and a bold variant, many have also a lean variant. In multiple-master fonts one can continuously vary this aspect of a font from hyper-lean to hyper-bold within the font (no need for parallel fonts). This setting does not alter the colour of the text and does not affect in-line images (like emoji) in the text, nor prestyled bold, italic or fractur characters, nor ‘line drawing’ (for terminal emulators) characters.

- **CSI 1**[**:***v*]**m** Bold font variant. Cancels lean and normal (and current bold variant). Variants (note that the separator is colon, *not* semicolon): Variants: **CSI 1:700m** bold (default for **CSI 1m**), **CSI 1:400m** normal weight (alias: **CSI 22m**), **CSI 1:600m** semibold (if available, otherwise bold), **CSI 1:800m** extra-bold (if available, otherwise bold). **CSI 1:300m** select lean/light weight (alias: **CSI 2m**). *v* can be a value between 1 and 1000 (inclusive) selecting a weight as in OpenType and CSS. *v*=0: do a *relative* change to "lighter" as in CSS '*font-weight: lighter*', *v*=1001: do a *relative* change to "bolder" as in CSS '*font-weight: bolder*'.

- **CSI 2**[**:***v*]**m** Lean/faint (light) font variant. Same as **CSI 1**[**:***v*]**m**, but the default for *v* is 300.
- **CSI 22**[**:***v*]**m** Normal weight (neither bold nor lean). Same as **CSI 1**[**:***v*]**m**, but the default for *v* is 400. 400 is also the initial (default) font weight.

Note: there is *NO CHANGE IN COLOUR* (or 'intensity') for these font weight changes, only change in weight. Fonts that are multiple-master do this within the font, otherwise there may be a font change to a related font. Contexts that are strictly fixed width might not implement the more "extreme" bolds, even if available in the fonts, since they usually need wider advance widths. This styling does not apply to inline images (e.g., emoji) or characters that are prestyled w.r.t. boldness/leanness or italic/Fraktur/double-struck. The value of the font weight does not change the width of "en" and "em", even though the advance width may change.

This styling is autoterminated at text component end (end of table cell, end of text items in math expression and vector graphics, *if* started in the subcomponent), but is inherited to text components as well as into superscript/subscript and small/petit caps.

8.3 ROMAN/ITALIC (or OBLIQUE) STYLE (extended with variants)

This styling does not affect inline images (like emoji) in the text, nor prestyled bold, italic or Fraktur characters, nor 'line drawing' (for terminal emulators) characters.

- **CSI 3**[**:***v*]**m** Italicized or oblique (−7° to −10°). Oblique (normally synthesised) if there is no associated italic version. Variants: **CSI 3:0m** alias for **CSI 23m**, default **:1**: **CSI 3:1m** italicized (if available, otherwise approx. −7° to −10° oblique), **CSI 3:1=m**: +7° to +10° oblique (normally synthesised) which is sometimes used in RTL scripts. This styling does not apply to inline images (e.g., emoji), most symbols (S* except Sc; does apply to P* except / and \) or characters that are prestyled w.r.t. boldness/leanness or italic/Fraktur/double-struck (e.g., MATHEMATICAL characters). Does not apply to Fraktur or other "calligraphic" fonts, if such a font is set, this setting also changes to the default font. The value of *v* can be interpreted as "−*v* times 10 degrees glyph slant", though the exact factor, 10 here, is implementation defined. A value of *v* between 0.7 and 1.0 uses an italic variant if available for the current font but modifies the inclination between more upright to more oblique, with 1 meaning "as is".

- **CSI 23**[**:***v*]**m** Roman/upright, i.e. not italicized/oblique (default), slant is 0° from upright when *v* is 0 (default). If the current font is a "calligraphic" font (like Fraktur), this setting also changes to the default (and non-calligraphic) font. Same variants as **CSI 3**[**:***v*]**m**, but the default variant is **0**. In addition, variant **1** is always interpreted as oblique (−7° to −10°, implementation defined), never as italic. Does not apply to characters that are prestyled w.r.t. italics/bold/Fraktur. Upright is also the initial (default) setting for this.

Note that the slant *direction* does ***not*** change by bidi level nor by paragraph direction setting for bidi. But the slant can be positive (by negative argument).

This styling is autoterminated at text component end (end of table cell, end of text items in math expression and vector graphics, *if* started in the subcomponent), but is inherited to text components as well as into superscript/subscript and small/petit caps.

8.4 FONT CHANGE (extended with variants; Fraktur is moved here)

One of the following two non-calligraphic fonts should be the default or initial font. Changing font does *not* change any currently set tabulation positions, even if the 'em' changes.

- **CSI 20m** Change to a predetermined calligraphic font (set). So-called "calligraphic" fonts are extra adorned fonts. This includes Fraktur (https://en.wikipedia.org/wiki/Fraktur). Italic/oblique does not apply to this kind of fonts, so "italics" is implicitly cancelled when changing to a calligraphic font. Which font (or font set with different weights) may be given by the implementation or may be settable in preferences or an **APC** control string. A calligraphic font cannot be the default font (not even by preference setting).
- **CSI 26**[:*v*]**m** Change to a predetermined proportional (non-calligraphic) font (set) (serif or sans-serif). Variants: **CSI 26:1m** serif, **CSI 26:2m** sans-serif. Which font (or font set with different weights, condensed/expanded, and italic or not) may be given by the implementation or may be settable in preferences (or an **APC** control string). **CSI 26:0m** resets to initial (default) font, serif, sans-serif, or "fixed" width.
- **CSI 50m** Change to a predetermined "fixed" width (non-calligraphic) font (set). Kerning is not applied. Which font (or font set with different weights, condensed/expanded, and italic or not) may be given by the implementation or may be settable in preferences or an **APC** control string (not specified in this proposal). Though called "fixed width", it will have up to three different widths (not counting text "objects" like math expressions, tables, etc.): 0-width (non-spacing diacritics, as well as some interpreted control strings; though the latter will not be handled by the font per se, but long before), narrow (most scripts), wide (double-width) for CJK, Hangul and emoji (should be per unicode.org/Public/UCD/latest/ucd/EastAsianWidth.txt). Note that some characters come as both emoji and non-emoji (may use variant selectors to indicate which), and non-emoji vs. emoji have different widths for the same character. (Fixed-width fonts are often the only option in a terminal emulator, and fixed-width font is often the assumption in terminal-oriented text editor programs. It is not a general requirement for terminal emulators, though.)
- For *y* in **10** to **19**, **CSI** *y***m**: Pick from a font palette of up to ten fonts (set permanently, as preferences or via **APC** control strings). Each index refers to a set of fonts, varying in italics and weight, if not multiple-master. E.g., **CSI 19;1m** (i.e., **CSI 19mCSI 1m**) will select the bold version of the font at index 19 (whether multiple-master or separate font file for the bold). For each such font set, there needs to be information whether it is "calligraphic" or not (for the interpretation of **CSI 3m** and **CSI 23m**). The FNT control sequence, **CSI** … **SP D,** is hereby deprecated.

Note: Xterm has an extension with which one can specify a font by name, a font to change to or change the font palette. It uses **OSC** but should use **APC.** (We will not copy that functionality below, but leave it as a "private" extension in Xterm, at least for now. Reason: we will not define *any* particular control string functionality in *this* proposal (just the "overall" syntax for control strings).)

Changing font does not in itself change the width of spaces (general category Zs) (except when changing between fixed-width and varying widths fonts). The advance widths of spaces are not to be directly read from the font (except for OGHAM SPACE MARK, which may have a glyph, and spaces whose widths correlate with the advance width of ASCII digit "0" (in "tabular" style) or ASCII punctuation "."/","). However, for fixed-width fonts the nominal advance width for SP and NBSP are 1 en. For variable-width ("proportional") fonts, the nominal advance width for SP and

NBSP is usually set to a somewhat smaller value (like 0,7 to 0,9 en, but is implementation defined, and may be settable as a user preference). Note in addition, that ADVANCEMENT MODIFICATION as well as LINE JUSTIFICATION (see below) may modify the width of instances of SP and NBSP.

This styling is autoterminated at text component end (end of table cell, end of text items in math expression and vector graphics, *if* started in the subcomponent), but is inherited to text components.

8.5 FONT SIZE CHANGE (new, replacing GSS, GSM)

In "original" ECMA-48, units for sizes and advances were set by a separate control sequence, SSU (SELECT SIZE UNIT). This design is severely flawed for several reasons (major reason: the unit is separated from the value and that can easily lead to very wrong interpreted values, especially if one is doing cut-and-paste of raw ECMA-48 styled strings, or similar kinds of substitutions). Replacements for "SSU dependent" control sequences are proposed below. But we do keep the unit codes from SSU, for backwards compatibility on that point. Zoom is applied after size determination. Changing font size or font magnification does *not* change the position any currently set tabulation positions, even if using the unit with code 0 (en); i.e. the tab position are *not* stored in terms of the unit 0 (en).

Unit codes (**0** to **8** from SSU, **9** is new; an implementation need not support all units, but **0**, **1**, **2**, **7**, and **9** are expected to be supported, compare CSS):

- **0** CHARACTER – 'en' (0,5 em) in the character progression direction (usually horizontally), even for double-width scripts (CJK); 'em' in the line progression direction (usually vertically) ('em' is also supported as a unit in CSS)
- **1** MILLIMETRE (this is called 'mm' in CSS, it is not necessarily exactly 1 mm)
- **2** COMPUTER DECIPOINT – 0,03528 mm (254/7200 mm) (this is called 'pt' in CSS)
- **3** DECIDIDOT – 0,03759 mm (10/266 mm)
- **4** MIL – 0,025 4 mm (254/10 000 mm)
- **5** BASIC MEASURING UNIT (BMU) – 0,02117 mm (254/12000 mm)
- **6** MICROMETRE – 0,001 mm
- **7** PIXEL – The smallest increment that can be specified in an (old!) display device (this is called 'px' in CSS, it is *not* necessarily one actual pixel in size, especially with modern high-resolution devices 1px may be two or more actual pixels)
- **8** DECIPOINT – 0,035 14 mm (35/996 mm)
- **9** default size (value parameter ignored) (the top "pushed" size w.r.t. **CSI ![**, see below; if none, it is the global default size, which may be settable as a user preference)

They should be interpreted in the same way as in HTML/CSS (extended for the units that are not in HTML/CSS), see https://www.w3.org/Style/Examples/007/units.en.html.

- **CSI 76**[**:***u***:***s*]**m** Font size. Unit *u* as per above. *s* can be negative with EQUAL SIGN (=) postfix, indicating negation (for an upside down mirrored glyph), *s* can be fractional with **?** as decimal marker, e.g. **CSI 76:2:14?5m**, for 14,5 points. **CSI 76:9m** (no size indicator) resets to default size, **CSI 76m** is approximately the same as **CSI 76:0:0?8m** (superscript/ subscript size, but no superscripting/subscripting). This replaces GSS (GRAPHIC SIZE SELECTION), which is here proposed not to be used nor implemented. SSU is also proposed not to be used/implemented, and the units are given in the SGR control code sequences instead. This replacement is due to several factors, because GSS was ill designed. a) The unit was set separately, which can cause major unintended rendition flaws. b) Even though there were several units associated with this, GSS did not allow for decimal values, so no (near accurate) conversion was possible, nor did it allow for negative values (which is of course optional to implement). c) Sizing is to be reset by **CSI 0m**, but GSS was not. Note that increasing the font size may temporarily increase the line spacing in order to accommodate the larger glyphs. The "em" unit should be glyph size based (e.g., based on Åg-height or similar glyph measure, and the adjustment may vary even over a single font), not just verbatim size from the font, so that glyphs appear size-wise similar for a given requested size. Changing the font size also changes the size of the en and em (approx. Åg height, *after* adjusting the declared font em) units. The Åg height (or similar) refers to the nominal size, *not* the superscript/subscript (subs/sups) size nor the small caps (c2sc/c2pc) size.

- **CSI 77:***a***:***b***m** Font magnification. *a* (vertical factor) and *b* (horizontal factor) can be fractional ('**?**' as decimal marker), and *b* can be negative ('**=**' as negation sign, postfix), e.g., **CSI 77:1:0?8m** for narrow (may use condensed/expanded axis in a multiple master font), **CSI 77:1:1m** resets. This setting is *not* accumulative. Arguments are direct factors, not percent. This replaces GSM (GRAPHIC SIZE MODIFICATION) which is here proposed not to be used. This replacement is due to several factors, because GSM was ill-designed. a) It was percentages but did not allow for decimal values. b) Font magnification is to be reset by **CSI 0m**, but GSM was not. This setting should take into account condensed/expanded variants in a multiple master font. Increasing the *vertical* height via magnification will "glue" the glyphs at cap height, making them expand below the (normal) baseline on horizontal lines *without* increasing the line spacing, whereas increasing the font size "glue" them to the baseline, growing the line spacing for the lines with increased font size. Changing font magnification does *not* change the size of the en and em units, but changes the glyph and space advance.

This styling is autoterminated at text component end (end of table cell, end of text items in math expression and vector graphics, *if* started in the subcomponent), but is inherited to text components.

8.6 UPPER/LOWER HALF GLYPHS (new, for ISCII support)

This control sequence is strongly not recommended. It is proposed *only* to make certain conversions from ISCII more direct (or in some aberrant (non-use-case) cases: possible). Xterm also has similar functionality, but whole line only and using **ESC #3** and **ESC #4** (are reset at eol).

- **CSI 113:***v***m** Show horizontally halved glyphs. Variants: **:0** show glyphs normally, **:1** show only the upper half of the glyphs (for combining sequences), cutting about half cap height and counting only half an em for these glyphs for line height calculations, **:2** show only the lower half of the glyphs (for combining sequences), moved up to just under the half glyphs of previous line (so that two lines with the same text, constant size, the upper half joins with the lower half). Note that this only does the "glyph halving", size change (often to double height) must be done separately (in contrast to ISCII and xterm).

This styling is autoterminated at text component end (end of table cell, end of text items in math expression and vector graphics, *if* started in the subcomponent) and at NLF, and is *not* inherited to text components.

For other use than ISCII and xterm compatibility (after conversion to using this ECMA-48 based extension), this styling is strongly deprecated.

8.7 RAISED/LOWERED TO FIRST INDEX POSITION (new, replacing PLU, PLD)

ECMA-48 has PLU (PARTIAL LINE UP/BACKWARD) and PLD (PARTIAL LINE DOWN/FORWARD) for making indices (if the character progression direction is horizontal), mimicking manual typewriters, moving the paper. They do have a bit of generality, and *nroff* in its days used such partial line movements to output mathematical expressions using typewriter-like terminals (but for mechanical reasons, the printed lines had to be rearranged so that the paper only moved in one direction and the paper passed through several times, with different type wheels).

That way of printing math expressions is not recommended. For math expressions, allowing for good typography and layout, we recommend the math expression representations presented in the separate proposal *A true plain text format for math expressions (and its XML compatible equivalent format)*, which is compatible with ECMA-48 based styling.

In addition, superscripting and subscripting (when not in a mathematical expression) are now often regarded as character styling. Indeed, OpenType fonts have dedicated styling "features" for superscripts and subscripts (not intended for math expressions) but can otherwise be achieved by relative sizing and relative placement of the "ordinary" glyphs.

The control sequences below will make characters display at the first index position, raised or lowered. This replaces PARTIAL LINE FORWARD and PARTIAL LINE BACKWARD codes for making indices. PLD/PLU do not do the size change normally associated with superscripts or subscripts.

Note that for some uses, like powers of units (like m^2), chemical numerical subscripts (like H_2O) as well as phonetic notation (like ^{h}), preformatted Unicode characters are *strongly* preferred over using the below styling to achieve the raised/lowered display. However, for unit superscripts and chemical notation with subscripts, even more preferable is to use math notation, if available to use. Preformatted Unicode characters are also sometimes preferred for certain abbreviations, like ordinals with letter superscripts, and also for other common abbreviations in (e.g.) French. Preformatted superscript/subscripts are not affected by the control sequences for superscript/subscript styling, thus retaining their preformatted style in this regard.

On the first instance after a "normal" character of a switch directly from superscript to subscript or vice versa, whether by these control sequences or preformatted superscript/subscript characters, the display position is moved back to after the last "normal" character. When ending, the display position is moved to after the longest of the two sequences. This way, one can apply superscript and subscript nicely together *after* a "normal" character. However, attempting to apply multiple raised or multiple lowered strings to the same "normal" character will have an implementation defined effect. For mathematical expressions represented in a way compatible with ECMA-48 based styling, see *A true plain text format for math expressions (and its XML compatible equivalent format)*.

If the character progression direction is vertical, the superscripts and subscripts apply *horizontally* (opposite the line progression direction) to the preceding "normal" character.

- CSI 56**:1**[**:***v*]**m** Raised to first superscript level and slightly smaller, about 80% current set font size. (May use the OpenType feature 'sups' if available.) Small/petit caps formatting has no effect on the superscripted text. Overline will temporarily skip; strike-through and underline are not affected (i.e. not moved but still displayed). This setting does not accumulate, i.e., one cannot achieve second level superscript or superscript of a subscript. Terminates lowered to first subscript level.

Variants: *v*=0 (default): no underline at superscript level, *v*=1: singly underlined at superscript level, *v*=2: doubly underlined at superscript level. These superscript underlines are not sensitive to the underline/overline/… colour setting but always follow the "foreground" colour setting.

- **CSI 56:1=m** Lowered to first subscript level and slightly smaller, about 80% current set font size. (May use the OpenType feature 'subs' if available.) Small/petit caps formatting has no effect on the subscripted text. Underline will temporarily skip; strike-through and overline are not affected (i.e. not moved but still displayed). Does not accumulate, i.e., one cannot achieve second level subscript or subscript of a superscript. Terminates raised to first superscript level.
- **CSI 56**[**:0**]**m** Not raised/lowered and back to the set font size. (Default.) This reset should be autoapplied at text component start and end, as well as at NLF, HT/HTJ and SP.

Superscripting and subscripting is a very common notation in mathematics. But those superscripts subscripts (that are part of math expressions) will need to use a different mechanism in order to fit with math expression layout. See details in *A true plain text format for math expressions (and its XML compatible equivalent format)*, 2024, https://github.com/kent-karlsson/control/blob/main/math-layout-controls-2024.pdf.

Text superscript/subscript is *autoterminated* at the *start* of a math expression, table, vector graphic, image, emoji or a hard line break (NLF), and also at the end of a component. There should be no automatic line break within a text superscript/subscript even if it contains spaces or punctuation that would otherwise allow automatic line break.

8.8 LIGATURES (new)

Modern fonts often support ligatures. However, fixed width fonts usually do not have ligatures, though some scripts do have required ligatures ('rlig' feature in OpenType). Proportional fonts should have required ligatures (if any in the scripts supported), and modern ligatures ('liga' feature in OpenType). Exactly which (modern) ligatures are supported depends on the font, but commonly (for the Latin script) fi, fj, fl, fb, fh, fk, ff, ffi, ffj, ffl, ffb, ffh, ffk would be covered by 'liga', but may cover more, like ft, fft, and even 'if', 'iff', 'jf' and tf, as well as fö, ffö, fä, ffä and others.

However, emoji "ligatures" of emoji sequences will not use this (neither the OpenType features, nor the below control sequences) but need to use another mechanism. Indeed, ligated emoji sequences are insensitive to bidi, and must be parsed out prior to bidi rearrangement.

- **CSI 57**[**:0**]**m** Reset to default w.r.t. ligatures. Fixed width font: same as **CSI 57:1m**. Proportional width font: same as **CSI 57:2m**. Note that for proportional fonts, kerning is always enabled.
- **CSI 57:1m** Only required ligatures (none in the Latin script). A required ligature (of two letters) may have the same width as one letter. OpenType feature 'rlig'.
- **CSI 57:2m** In addition to required ligatures, use also modern ligatures (according to the font). For a fixed width font, this is usually the same as **CSI 57:1m**. OpenType features 'rlig' and 'liga'; 'liga' ligatures should not incur any width changes, or very small ones.

- **CSI 57:*v*m**, *v* = 3 or greater. Reserved for future use; historical/‘hlig’ ligatures (include in **CSI 57:2m**?); discretionary/‘dlig’: mostly for [acegmnopqrs][bfhklt] where one can connect the letter pair by an arc (ct; which used to be popular); ‘clig’/‘calt’ contextual alternatives.

8.9 SMALL CAPS (new)

Small caps style is often the preferred style for abbreviations/acronyms in uppercase. Note that for phonetic notation, preformatted Unicode characters are strongly preferred. This styling does not apply to letters that are already small caps. Small caps can be simulated by using a smaller font size. Exceptionally, lowercase as small caps are sometimes used in headings or lead text. Note though that the mapping to uppercase is sometimes language dependent, for instance for German and Turkish. These styling variants only apply to cased alphabets, and has no effect on letters from uncased alphabets/ideographs/etc.

- **CSI 66**[**:0**]**m** Normal uppercase and lowercase rendering (default). Resets to previous **CSI 67:*v*m** that was in effect before a “recent” **CSI 66:1m** or **CSI 66:1m** (any nesting with **CSI 66…** and **CSI 67…** beyond what is here specified has implementation defined effect).
- **CSI 66:1m** Uppercase letters rendered in smaller than normal size (but may be slightly greater than x-height), small caps. Lowercase letters are not affected, but *ASCII digits and “&” should be affected* (if not in the font, simulate). OpenType feature 'c2sc' (simulate if not available). This is *regularly* used for uppercase abbreviations/acronyms and other stretches of uppercase letters. Preformatted small caps letters are already in this format and are unaffected by this setting, as are preformatted and explicit superscript/subscript.
- **CSI 66:2m** Uppercase letters rendered in even smaller than normal size (approximately x-height), also called petit caps. OpenType feature 'c2pc'. Sometimes a distinction is made between small and petit caps. Lowercase letters are not affected, but *ASCII digits and ‘&’ should be affected*. If petit caps are not available, this selects small caps, maybe simulated. Preformatted small caps letters are unaffected by this setting, as are preformatted and explicit superscript/subscript.
- **CSI 66:666m** Lowercase letters rendered as the petit caps of their uppercase. Original uppercase letters are not affected. Digits are not affected. OpenType feature 'pcap'.
- **CSI 66:999m** Lowercase letters rendered as the small caps of their uppercase. Original uppercase letters are not affected. Digits are not affected. OpenType feature 'smcp'. This is *sometimes* used for headings/titles/captions and even first paragraph lead texts.

Note that mapping to uppercase is sometimes language dependent, and this display mode might not respect that (implementation defined); however LATIN SMALL LETTER SHARP S is always uppercase mapped to LATIN CAPITAL LETTER SHARP S, and accent marks are never dropped when mapping to uppercase. Small caps and petit caps from lowercase can be seen as violating Unicode character identity.

While uppercase to small/petit caps and lowercase to small/petit caps in principle could be combined (turning all into small/petit caps), that is not supported here. Likewise, map to (normal size) uppercase *as a display style* (turning all into uppercase) is not supported here. Nor is mapping uppercase to lowercase *as a display style* supported here.

Text small caps/petit caps is *autoterminated* at the start of a: math expression, table, vector graphic, image, emoji, and also at the end of such a component as well as at a hard line break (NLF, including VT/LS).

8.10 UPPERCASE/LOWERCASE & FIXED-WIDTH/PROPORTIONAL DIGITS (new)

Lowercase digits are often preferred in running text (which is mostly lowercase). For tabular-like contexts or arithmetic expressions, fixed width digits are preferred. These styles apply only to the (narrow) digits 0 to 9, not digits in other scripts, nor fullwidth 0 to 9.

- **CSI 67**[**:0**]**m** Uppercase fixed width digits (default). OpenType feature 'lnum' and, if needed, 'tnum'; 'tnum' is not needed for fonts that are (generally) fixed width.
- **CSI 67:1m** Lowercase fixed width digits. Sometimes called "old-style" digits/figures, despite modern use. If available in the font, OpenType feature 'onum' and, if needed, 'tnum'. (No effect if lowercase digits are not available in the font.)
- **CSI 67:2m** Uppercase proportional width digits. Has effect only if available in the font. OpenType features 'pnum' and 'lnum'. For fixed width fonts: same as **CSI 67:0m**.
- **CSI 67:3m** Lowercase proportional width digits. Sometimes called "old-style" digits/figures, despite modern use. Has effect if available in the font. OpenType features 'onum' and 'pnum'. For fixed width fonts: same as **CSI 67:1m**.
- **CSI 67:99m** Uppercase fixed width digits, where the (ASCII) 0 has a middle dot or an internal diagonal stroke. OpenType feature 'zero' with 'lnum' and, if needed, 'tnum'. This style was originally (long before OpenType) for computer terminals (and printers), but now used more for effect and rarely used in terminal emulators. Same as **CSI 67:0m**, if this variant is not available, but some fonts may have this as the only variant for 0.

Note that 'lnum' (uppercase) and 'onum' (lowercase) are contradictory, as are 'tnum' (fixed width) and 'pnum' (proportional width). Other combinations are ok. **CSI 66:1m** and **CSI 66:2m** also should affect ASCII digits, rendering them as small/petit caps, regardless of **CSI 67:***v***m**.

This styling is *autoterminated* at text component end (end of table cell, end of text items in math expression and vector graphics, *if* started in the subcomponent), but is inherited to text components.

8.11 LINE INDENTS AND JUSTIFICATION (new, replacing JFY, QUAD, SLH, SLL)

Line indents can be used for first line intent for "ordinary" paragraphs, and for non-first line indents mostly for bulleted/numbered lists. For bulleted/numbered lists, the bullet or the number can be in a "tab field" and have an HTJ (CHARACTER TABULATION WITH JUSTIFICATION, U+0089 or, as a character reference, **ESC I**) just after the bullet/number part, and then a HT to the tab position set at the same position as the non-start line indent (<HTJ,HT>).

The line indent control sequences should occur only at the beginning of a paragraph. At other positions, the result is implementation defined.

Line indents are *not* reset by **CSI 0m** (but are reset by RESET TO INITIAL STATE, **ESC c**, but like **CSI 0m**, it is not a good fit for a document format, and thus might not be implemented for a text document format, though useful for terminal emulators).

SLH (SET LINE HOME), SLL (SET LINE LIMIT), SPH (SET PAGE HOME) and SPL (SET PAGE LIMIT) should not be used since they are ill-defined. However, this proposal does not (in this edition) propose any replacements for SPH and SPL.

- **CSI 69:***u***:***v*[**:***s*]**m** First line indent of paragraph if *s* is 1 (default), after LS/VT/CRVT if *s* is 2. Distance after (in the character progression direction) *default* BOL (beginning of line) position (as set by preference setting or by context (such as window size, or cell size)). On left side for LTR, on right side for RTL, on top for vertical lines. Does not accumulate. Unit *u* as per SSU units (0 to 8), 9 for reset to default (value ignored and can be omitted). *v* is positive (or zero), can be fractional. If *s* = 2 variant has not been explicitly set after an *s* = 1 variant setting, the *s* = 2 setting inherits the *s* = 1 setting from **CSI 70:***u***:***v*[**:***s*]**m.**

- **CSI 70:***u***:***v*[**:***s*]**m** Non-first after automatic line break) line of paragraph if *s* is 1 (default), after LS/VT/CRVT if *s* is 2. Distance after *default* BOL position (as set by preference setting or by context (such as window size, or cell size)). On left side for LTR, on right side for RTL, on top for vertical lines. Does not accumulate. Unit *u* as per SSU units (0 to 8), 9 for reset to default (value ignored and can be omitted). If *s* = 1 (default), this also sets two tab stops that "belong" to the **CSI 70:***u***:***v***:1m** (*s* = 1) setting: one at the position given, and one 1 en (current font and size; additional spacing can be achieved by using various space characters) prior (in the character progression direction) to that position. These two tab stops are not altered via HTSA2 (see below). Default tab stops before (in the character progression direction) these two tab stops are ignored when "tabbing". *v* is positive (or zero), can be fractional. (Nit: hanging indents can be useful for terminal emulators as well.) If an *s* = 2 variant is not set after an *s* = 1 variant setting,
  the *s* = 2 setting inherits the *s* = 1 setting from **CSI 70:***u***:***v***:1m**.

- **CSI 71:***u***:***v*[**:***j*]**m** EOL (end of line) line indent. Distance before (in the character progression direction) *default* EOL position (as set by preference setting or by context (such as window size, or table cell size)). On right side for LTR paragraph direction, on left side for RTL paragraph direction, on bottom for vertical lines. Does not accumulate. Unit *u* as per SSU units (0 to 8), 9 for reset to default (remaining two parameters are then ignored and can be omitted). *v* is positive (or zero), can be fractional.
  j = 1 (default): Flush the lines so that they begin as set by **CSI 69:***u***:***v*[**:***s*]**m**/**CSI 70:***u***:***v*[**:***s*]**m**.
  j = 2: Flush lines (the part after beginning of line, or if tab stops are used then after last **HT**/**HTJ** in the line; preceding part is treated as for *j* = 1) so that they end as set by **CSI 71:***u***:***v*[**:***j*]**m** (space/tab characters, at end of the line are not counted).
  j = 3: Centre lines between the positions set by **CSI 70:***u***:***v*[**:***s*]**m** and **CSI 71:***u***:***v*[**:***j*]**m** (spaces/tabs at end of line are not counted); if there is any HT/HTJ in the line then the centring is done between "its" tab stop and position set by **CSI 71:***u***:***v*[**:***j*]**m** (preceding part is treated as for *j* = 1). (Note that **CSI 69:***u***:***v*[**:***s*]**m** setting is not used.)
  j = 4 (replacing JFY): Stretch lines *that are not at end of paragraph nor ends with (CR)VT/LS/(CR)FF* (otherwise, treated as for *j* = 1) between (the nearest of) BOL (**CSI 69:***u***:***v*[**:***s*]**m** and **CSI 70:***u***:***v*[**:***s*]**m**); if there is any HT/HTJ in the line then the centring is done between "its" tab stop and position set by **CSI 71:***u***:***v*[**:***j*]**m** (preceding part is treated as for *j* = 1) (spaces/tabs at end of line are not counted) to fill up the line space by widening SPs, NBSPs, IDSPs, though that should be avoided for spaces before/after words in certain cursive scripts (e.g. Arabic) where the words are instead lengthened by adding ARABIC TATWEEL to some of the words.
  j having a negative value, **1=**, **2=**, **3=**, **4=**: Same as for the corresponding positive values, but no automatic line breaking.
  j = 5 or *j* = 6 (numerical alignment): No automatic line breaking. First align VT/LS/… separated lines in each line's (of the paragraph) final tab field (all lines should use the same final tab field) on first occurrence of COMMA, FULLWIDTH COMMA or ARABIC DECIMAL SEPARATOR (*j*=5) or to FULL STOP or FULLWIDTH FULL STOP (*j*=6) (align with end of line if no occurrence of those characters in the line), then align the entire paragraph to end of line as set by **CSI 71:***u***:***v*[**:***j*]**m** (assuming that the line contents at all fits). This is

intended for numeric alignment of (LTR) numerals, one numeral in the last used (and same) tab field per LS/VT separated line of the paragraph. See the section below on tables for aligning numerals in table columns, and the math expression proposal for aligning numerals in math matrix expressions.

The behaviour for line content that cannot fit in the line is implementation defined, but should normally use automatic line breaking.

Except for *j* = 1, if there are HT/HTJ characters in the line, only the part after the last HT/HTJ character in the displayed line is affected by the adjustment from that HT/HTJ character's "target" position, which is used as *temporary* BOL position, to EOL position. Note that the SPD(2) setting affect where a text line starts and ends.

This styling is *autoterminated* at text component end (end of table cell, end of text items in math expression and vector graphics) *if* started in the subcomponent, and is inherited to text subcomponents (like table cells).

8.12 LINE SPACING (new, replacing SLS, SVS, SPI)

The line progression direction "movement" when there is a line break. Default: about 1,3 em.

- **CSI 72**[**:***u***:***s*[**:***sb*[**:***sa*]]]**m** Line spacing. *u* unit as per SSU (0 is em, 1 to 8 per SSU, 9 resets to default 1,2 em (counting the largest font size in the line) line spacing regardless of *s*, which then can be omitted), 9 is default for *u*. However, objects (such as tables, vector graphics, math expressions) are counted by their real size in the line progression direction, they are also centred in the line space (in the line progression direction), unless the object specifies differently (e.g., math expressions are aligned on the vertical math centre, which is usually not the true vertical centre). *s* can be fractional (e.g., **CSI 72:0:1?5m** for 1,5 em line spacing). This replaces SLS (SET LINE SPACING), SVS (SELECT LINE SPACING), SPI (SPACING INCREMENT) which are proposed not to be used. If the unit is 'em', it is the em (after adjustment, so Åg height really) of the largest glyph (considering font sizes) in the displayed line, not counting math expressions, tables, vector graphics or images in the line (which may give extra line spacing). *sb* (reusing the unit *u*), default **0**, is the extra spacing before a paragraph (in the line progression direction), *sa* (reusing the unit *u*), default *sb*, is the extra spacing after a paragraph. Extra spacing before a paragraph is not issued at the beginning of page (but is issued at the beginning of a table cell), extra spacing after a paragraph is not issued at end of page (but is issued at the end of a table cell). When extra space after a paragraph is adjacent to extra space before a paragraph, the max of the two is used, not the sum.

Line spacing is inherited into table cells. A line spacing control sequence set in a table cell is automatically reset at end of cell. Note that line spacing is *not* reset to default by **CSI 0m**, only by another line spacing control (**CSI 72m** resets line spacing to default), or by RESET TO INITIAL STATE (**ESC c**) (for terminal emulators, not recommended for document formats).

8.13 ADVANCEMENT MODIFICATION (new, replacing SACS, SDCS, SSW)

Not recommended, but extra letter separation has been used for emphasis, esp. in German Fraktur, but also in other instances.

- **CSI 110**[**:***z*]**m** Advancement modification (times **CSI 111:***x***m** (spaces only) and **CSI 77:***a***:***b***m** modification) in the character progression direction after kerning. *z* can be fractional (**?** as decimal marker), affects spaces as well, **CSI 110:1m** resets (factor = **1**; short **CSI 110m**), no accumulation. E.g., **CSI 110:1?3m**abc**CSI 110:1m**d sets “abcd” with 1,3 times the nominal advance width of each letter, “stretching” the word. Factors other than 1 may break up ligatures and cursive joins, but for Arabic ‘kashida’ strokes may be inserted. Factors less than 1 should *not* be used, as that may result in (partial) overtyping with implementation defined display results. (As an extreme example, **CSI 110:0m**abc**CSI 110:1m**d sets “abcd” all at the ‘same’ position.) Changing the advance does *not* change the “en” and “em” unit sizes, nor tab stops. This replaces SACS and SDCS which are here proposed not to be used. The latter depended on separately set unit and were not reset by **CSI 0m**, were additive, not multiplicative, and were reset by line breaks.

- **CSI 111**[**:***x*]**m** Space (Unicode general category Zs characters) advancement modification (times **CSI 110:***z***m** and **CSI 77:***a***:***b***m** modification), in the character progression direction; this is before stretching SP, NBSP, IDSP for line justification reasons. *x* can be fractional (**?** as decimal marker); **CSI 111:1m** resets (factor = **1**; short: **CSI 111m**), no accumulation. Note that if the line is justified (**CSI 71:***u***:***v***:3m**), some spaces may be “stretched” a bit more than given here (but never less), due to line justification. Values less than 1 should not be used. Changing the space advance factor does *not* change the “en” and “em” unit sizes, nor tab stops. This replaces SSW which is here proposed not to be used. The latter depended on separately set unit and was not reset by **CSI 0m**, was additive, not multiplicative, and was reset by line breaks.

This styling is *autoterminated* at text component end (end of table cell, end of text items in math expression and vector graphics) *if* started in the subcomponent, and is *not* inherited to text subcomponents (like table cells).

8.14 UNDERLINE (extended with variants)

ECMA-48 does not allow for several different types of underlines on the same piece of text, and even if one were to allow it, it would not be possible to have differently coloured underlines on the same piece of text (but see the section on UI underlines below, which are *not* part of the text styling, but part of the UI). Similarly for overline and strike-through. The underline is drawn just below the baseline. The underline should skip descenders, like those in “jgpqy” as well as letters with diacritics below. But do not skip LOW LINE, which nowadays is usually displayed on the baseline, not below the baseline, even though its origin in typewriters was for underlining.

- **CSI 4**[**:***v*]**m** Singly underlined. Variants: **CSI 4:0m** not underlined (same as **CSI 24m**), **CSI 4:1m** solid (medium, default), **CSI 4:2m** double thin solid lines (same as **CSI 21m**), **CSI 4:3m** wavy/curly, **CSI 4:4m** dotted, **CSI 4:5m** dashed (medium), **CSI 4:6m** double wavy, **CSI 4:7m** thin solid, **CSI 4:8m** bold solid, **CSI 4:9m** bold wavy, **CSI 4:10m** thin dashed, **CSI 4:11m** bold dashed, **CSI 4:12m** double thin dashed. Each variant of **CSI 4m** cancels current **CSI 4m**, if any, and cancels **CSI 21m**. Thin and thick is relative to current font size.
- **CSI 21**[**:***v*]**m** Doubly underlined. **CSI 21m** cancels **CSI 4m**, if any. Same as **CSI 4**[**:***v*]**m**, but with different default (**:2**).
- **CSI 24**[**:***v*]**m** Not underlined. Almost the same as **CSI 4**[**:***v*]**m**, but with different default (**:0**). However, if the underlining is *initiated* via this control sequence (i.e., non-zero *v*), the underline is a bit lower so that it goes under common descenders (for g, j, etc.) and there is no gap in the underlining for such letters, letters with single diacritic below, nor for lowercase digits, nor for “simple” subscripts. (Compare **CSI 51m**.)

The underline is drawn in the character progression direction and is on the line progression direction side of the text line. Compare HTML underline (`<span style="text-decoration: underline">`). The underlining "follows" the baseline (not text superscript/subscript) of the text (note in particular for Indic scripts, where the glyphs align on a high line) in case of size changes. However, there is no underlining of math expressions (**CSI 24**[**:***v*]**m**, *v* > 0: that go below first subscript level), vector graphics, tables or images. Adding underline does not change the spacing in any direction. Underline is *not* inherited into math expressions, table cells or vector graphics texts.

Negating the variant indicator (e.g., **1=**), will slightly decrease the size of the glyphs that are underlined, while anchored at cap height, to make space for an underline that is *just above* the original baseline (instead of just under it). This special treatment of negated variant indicators is only for underline, not overline nor strike-through. This way of underlining is sometimes used for abbreviations, like an underlined "o" for "och" ("and", used instead of &), **CSI 4:1=m**o**CSI 24m**, but may of course also be used for stylistic reasons. For underlined superscripts, see section 8.7, which are similar (but smaller and higher), are sometimes used for abbreviations, but then usually for suffix inflections.

8.15 OVERLINE (extended with variants)

The overline is drawn slightly above cap height. Used for some math notations, like decimal sequence repetition for rational numbers, for displaying arithmetic calculations (sum, …, long division; use **CSI 55**[**:***v*[**:***w*]]**m** to avoid unexpected gaps).

- **CSI 53**[**:***v*]**m** Overlined. Same variants as **CSI 4m**, default is **:1**, medium solid line. Each **CSI 53**[**:***v*]**m** cancels current **CSI 53**[**:***v*]**m**, if any. Gap for capital letters with diacritic above.
- **CSI 55**[**:***v*[**:***w*]]**m** Not overlined. Same as **CSI 53**[**:***v*]**m**, but with different default (**:0**). If overlining is initiated via this control sequence (i.e. non-zero *v*), the overline is a bit higher so that it goes over capital letter with single accent mark above (for Å, Ö, etc.), and no gap in the overlining for such letters, nor for "simple" superscripts (text or math expression).
  w, default value **0** (no "extra" line), having the value **1** indicates that a straight line should be drawn in the *line* progression direction (i.e. usually vertical) between the (would be) position of an underline (**CSI 24**[**:***v*]**m**) and the (would be) position of the overline (**CSI 55**[**:***v*[**:***w*]]**m**); compare **CSI 51**[**:***v*[**:***r*]]**m**. With *w*=**2**, the line is curved like a ')', with *w*=**3**, the line is curved like a '('. This "extra" line functionality is intended for various notational conventions for *long division* (but not excluding other uses).

The overline is drawn in the character progression direction and is on the opposite side of the line progression direction side of the text line. Compare HTML overline (`<span style="text-decoration: overline">`). The overlining "follows" the (not text superscript/subscript) capsline of the text in case of size changes. However, there is no overlining of math expressions (**CSI 55**[**:***v*[**:***w*]]**m**, *v* > 0: that go above first superscript level), vector graphics, tables nor images. Adding an overline does not change the spacing in any direction. Overline is *not* inherited into math expressions, table cells or vector graphics texts.

8.16 STRIKE-THROUGH (extended with variants)

Strike-through is commonly used to mark text that is to be deleted or otherwise outdated, or wrong, but still shown for comparison with new text (that may be marked in some way), or just marking as closed (e.g., issue is closed, proposal rejected, or similar).

- **CSI 9**[**:***v*]**m** Strike-through. Same variants as **CSI 4m**, default is **:1** (*v*=1), medium solid line. Each **CSI 9**[**:***v*]**m** cancels current **CSI 9**[**:***v*]**m**, if any.

- **CSI 29**[**:***v*]**m** Not strike-through. Same as **CSI 9**[**:***v*]**m** but with different default (**:0**).

The strike-through line is drawn in the character progression direction, at a suitable mid-text-line position, **CSI 56:…m** superscripts/subscripts get the cross-out at the "normal" character position's midline. Compare HTML line-through (`<span style="text-decoration: line-through">`). Strike-through *is* inherited into math expressions, table cells, vector graphics texts.

8.17 OVERSTRUCK TEXT GRAPHEMES (new)

In some contexts (like displaying arithmetic calculations), one may want to do a special kind of overstrike. One that applies to each text grapheme (e.g., combining sequence, but more general, esp. for Indic scripts).

- **CSI 116**[**:***v*]**m** Start overstriking text graphemes. *v* indicates the kind of overstrike: **:1** for / overstrike, **:2** for \ overstrike, **:3** for ↗ overstrike, **:4** for ↘ overstrike, **:5** for ↖ overstrike, **:6** for ↙ overstrike, **:7** for a crossed over (╳) overstrike, **:8** for a | overstrike, **:9** for a | with / (double) overstrike, **:10** for a | with \ (double) overstrike. **CSI 116:0m** (default) for stopping overstrike of text graphemes.

Graphemewise overstriking *is* inherited into math expressions, table cells and vector graphics texts.

8.18 EMPHASIS MARKING FOR CJK (extended with variants)

Traditionally, CJK does not use bold or italic/oblique but other ways of emphasising; and the "underline" is different (CJK glyphs have a lower baseline than non-CJK characters). Compare CSS (https://developer.mozilla.org/en-US/docs/Web/CSS/text-emphasis-style). Proposed here are some extensions to ECMA-48, to handle some variants that CSS covers. All of these should cancel "western" underline/overline, but not cancel strike-through/crossed-out.

- **CSI 60**[**:***v*]**m** CJK underline (on the right side if vertical character progression direction). Same variants as **CSI 4m**, default is **:1**, medium solid line. (Negative variant value can be used for overline/left side line.)
- **CSI 61**[**:***v*]**m** CJK double underline (on the right side if vertical character progression direction). Same as **CSI 60**[**:***v*]**m**, but different default (**:2**). (Negative variant value can be used for overline/left side line.)
- **CSI 62**[**:***v*]**m** CJK overline (on the left side if vertical character progression direction). Same variants as **CSI 4m**, default is **:1**, medium solid line. (Negative variant value can be used for underline/right side line.)
- **CSI 63**[**:***v*]**m** CJK double overline (on the left side if vertical character progression direction). Same as **CSI 62**[**:***v*]**m**, but different default (**:2**). (Negative variant value can be used for underline/right side line.)
- **CSI 64**[**:***v*]**m** CJK stress marks (dot placed under/over (right/left side if vertical writing)). Variants: **CSI 64:0m** no CJK stress mark or line (same as **CSI 65m**), **CSI 64:1m** filled dot (●) under/right each CJK combining sequence (default), **CSI 64:1=m** filled dot over/left, **CSI 64:2m** filled sesame (﹅) under/right, **CSI 64:2=m** filled sesame over/left, **CSI 64:3m** hollow dot (○) under/right, **CSI 64:3=m** hollow dot over/left, **CSI 64:4m** hollow sesame (﹆)

under/right, **CSI 64:4=m** hollow sesame over/left. Ruby text (see below, PTX) should be placed “outside” of these marks.

- **CSI 65m** Cancel the effect of the renditions established by parameter values 60 to 64. Each of 60 to 64 primary parameter values should cancel any of the five; including “itself” since the variant may change (solid, double, wavy, dotted, dashed; position; dot/sesame, filled/hollow).

Each of the six above cancel the “western” underline and overline, but not strike-through, nor framing. “Western” underline/overline cancel the CJK emphasis marks, but not framing. These emphasis marks are *not* inherited into math expressions, table cells or vector graphics texts.

8.19 FRAMING TEXT EMPHASIS (clarified and extended with variants)

Put an inline frame around a span of text. This styling is a bit like old Egyptian cartouches.

- **CSI 51**[**:***v*[**:***r*]]**m** Framed string. At line break, the framing is terminated (no line at EOL side) and restarted (no line at BOL side) on the new line. Cancelled by **CSI 51**[**:***v*[**:***r*]]**m**, **CSI 52**[**:***v*[**:***r*]]**m**, **CSI 54**[**:***v*[**:***r*]]**m** (which all may also start another framing). Line above is just above singly accented capital letters, like Å, É. Line below is just below descenders, for letters like “jgpqy”. Same variants for *v* as **CSI 4m**, default variant is **:1**, medium solid line. *r*, default **0**, gives corner rounding; **:0** is non-rounded corners, **:1** is for 0,25 en radius rounded corners, **:2** is for 0,5 en radius rounded corners, **:3** is for 0,75 en radius rounded corners **:4** is for 1 en radius rounded corners. If background colour is set at the same positions as encirclement, the colour change ideally follows the encirclement. If there is a font size change within a framed string, the effect is implementation defined.
- **CSI 52**[**:***v*[**:***r*]]**m** Encircled string. Same as **CSI 51**[**:***v*[**:***r*]]**m**, but the default for *r* is **4**.
- **CSI 54**[**:***v*[**:***r*]]**m** Not framed, not encircled. Same as **CSI 51**[**:***v*[**:***r*]]**m** and **CSI 52**[**:***v*[**:***r*]]**m**, but the default for *v* is **0**.

These are useful not only for Egyptian cartouche-like effect (which is an unusual use for these, but handy for explaining this), but also for such things like prompts in terminal emulators or grouping “special” letter sequences, or name labels. Compare framed “span”s in HTML (`<span style="border: solid">`). The framing border follows the *maximum* ascender/descender in the line of text (after automatic line breaking) including math expressions, tables, vector graphics and images, which are *not* skipped by the framing. Adding framing does not change the spacing in any direction. Framing is *not* inherited into table cells, math expressions (text parts), or vector graphics texts, but is autoterminated at the end of a subcomponent *if* started in the subcomponent.

8.20 COLOURING (extended and clarified, ‘named’ colours have fix colour)

The original ECMA-48 assumed only two “colour planes”, background and foreground. Modern text applications (used with common higher-level protocols for document formatting) have several more planes. Also, some uses of ECMA-48 have already, in existing implementations, been extended with more colour planes, for instance a separate colour plane for underlines.

User interface display of ECMA-48 styled text is here proposed to have the following logical graphic “planes”, from “bottom” to “top”. However, an implementation need not work explicitly with colour planes, and there may be more logical colour planes than described here.

Colour values should be in the interval 0—255, negative values interpreted as 0 and larger values interpreted as 255. A value is then rounded down to an available value on the device.

0) Anything that is behind the window (like other windows and desktop background). Usually this is not seen at all (within the window in question) due to opaque colours on the higher planes, but some applications currently do allow transparency so that what is behind the window can be seen.

0.5) TV/video image; for Teletext-like applications.

1) Background fill colour or image. This may be (partially) transparent, even though it usually isn't. May be settable in preferences.

1.5) Block background colours. Settable via FRAMED BLOCK, **CSI 114:…m** (see below, section 8.22). It is actually several planes if PTX table rows are used (see below, section 10).

2) Text background colour or better called text highlight colour if the default text background colour (plane 2) is fully transparent, which is common nowadays. Filling up the line height for the *full* line, even if the line height is varying within the line (usually horizontal but may be vertical for CJK and Mongolian script); from below descenders and up; or, for vertical text: the full width of the vertical line. Default should be fully transparent (unless there are only two colour planes). The default may be settable in preferences. Colour can be set by **CSI 48**:….**m**. A *negative* transparency for the colours on planes 4 to 6 may "punch trough" this colour plane; e.g. **CSI 43;38:2::0:0:0:255=m** will make a "gold" background "punched through" by the text, making the text show the colours (or image) of planes 0–1.5.

2.5) Text selection colours. Used for marking text as "selected". Fully transparent for parts that are not selected. Some systems may have a separate "search match highlight" level. The colours for these are set by the system but may be settable in preferences. For a more ECMA-48 like solution, they can be set by control sequences proposed in the "UI text marking" section below; however, such UI text marking control sequences should never be part of a document.

3 and 3.5) Shadow planes. For character "shadows". Colours set by **CSI 8***z*:….**m** (there can be multiple shadows, with different colours, all-around shadow is on plane 3.5). This kind of shadow is supported in HTML/CSS. Text decoration, like underlines, associated with shadowed text, should then also get a shadow. Where there is no shadow, these planes are fully transparent.

4) Emphasis marks (CJK emphasis marks, the *lines* for underline/overline). Defaults to follow text foreground colour. Colour can be set by **CSI 58**:….**m**.

5) Text foreground colour (colour for the character glyphs). Usually defaults to black colour. Default may be settable in preferences. Apart from transparency (including blinking and concealed (transparent) characters) and negative image, the colour setting does not affect emoji (or other inline images/graphics, if such are permitted). Colour can be set by **CSI 38**:….**m**.

6) Text "decoration" (the *lines* for strike-out/overstrike, and for encircled/framed). Defaults to follow text foreground colour. Colour can be set by **CSI 68**:….**m**.

6.5) Maybe multiple colouring planes for spell and style checking underlines. Can be set by control sequences proposed in the "UI text marking" section below; such sequences should never be part of a document.

7, …) Some systems allow for (temporary) popups (with styled text) overlaying the "real" text. From the point of view of this document, that is part of the UI, not the document display itself.

Implementations are not required to support all planes. E.g., an implementation may choose to omit the shadow plane and the text "decoration" plane. But the text background plane and the text foreground plane are required. Colour settings are inherited into table cells, math expressions (all parts), and vector graphics texts, but is autoterminated at the end of a subcomponent *if* started in the subcomponent.

Colours (red, green, blue, transparency for the actual colours (0 is fully opaque, 255 is fully transparent), cyan, magenta, yellow, black) are given as integer values 0 to 255. To specify more than 8 bits per colour per pixel: use ***nnn>mmm***, where ***nnn*** is between 0 and 255 inclusive, ***mmm*** is between 0 and 255 inclusive, and ***nnn*** alone is short for ***nnn>nnn***, i.e., the least significant 8 bits part is a copy of the most significant 8 bits. If the hardware supports, e.g., 10 bits per colour per pixel, then the least significant 6 bits are ignored. Similarly for 11 bits per each RGBT (or CMYK) colour component, etc., including for 8 or fewer bits per each RGBT colour component. Each colour parameter below may be of the form ***nnn>mmm***, but we will only use ***nnn*** in the exposition here. The **>** may be regarded as a base 256 decimal marker.

8.20.1 Text background colour (full colour as basis, clarified and extended)

Text background colour, or better referred to as text highlight colour when using more than two colour planes, and the default colour for this is fully transparent. For the allowed values for *r*, *g*, *b*, *c*, *m*, *y*, *k,* see above regarding values. 0 is the default value for transparency, i.e., fully opaque. Unless it is negative, the parameter *i* is ignored (and can be omitted, but not the **:** separators), so is the parameter *a*.

With just two planes this is the colour behind the text. With more colour planes this is the text highlight colour (preferably with a default that is fully transparent). Usage hint: colours should be selected so that there is sufficient contrast between (total of) background colour, and foreground colours (text, underlines/ overlines, string framing).

Any display or print colour calibration adjustments are external to any ECMA-48 based text.

- **CSI 48:0m** Reset background colour to default text background/highlight colour. The 0 can be omitted (**CSI 48m**). Same as **CSI 49m**. (Default is taken from top of style stack, if style stacking is implemented; and is the inherited setting in case of inheritance.)
- **CSI 48:1m** Fully transparent text background/highlight (i.e. no highlight, the fill background is seen behind the text). Short for **CSI 48:2::0:0:0:255m**. Recommended default.
- **CSI 48:2:[*i*]:*r*:*g*:*b*[:*t*[:*a*:0]]m** Text background colour as RGB(T) (the separator here is colon).
- **CSI 48:3:[*i*]:*c*:*m*:*y*[:*t*[:*a*:0]]m** Text background colour as CMY(T) (the separator here is colon).
- **CSI 48:4:[*i*]:*c*:*m*:*y*:*k*[:*a*:0]m** Text background colour as CMYK (the separator here is colon).
- **CSI 48:5:*p*[:*t*]m** Text background/highlight colour from colour palette. *p* from 0 and up. Size of palette is implementation defined. Palette colours may be fixed by the implementation or be (partially) settable in preferences. If settable, various colour themes may be settable for the palette. There is also a possibility to set colour palette values via control sequences. If a transparency value, *t*, is given in this control sequence, that overrides the transparency value from the palette.

- **CSI 48:6m** Copy current foreground colour to background colour. Not recommended. Useful (only) for making conversion from Teletext more convenient. Deprecated.
- **CSI 49m** Reset to default text background/highlight colour (may be settable in preferences); for subcomponents, resets to the inherited background/highlight colour.

The text background/highlight colour is inherited to text subcomponents: to table cells, to math expressions (all parts), to vector graphics (text/label parts; perhaps with the condition that the text/label parts are well integrated with ECMA-48 formatting).

8.20.2 Text foreground colour (full colour as basis, clarified and extended)

Each of *r*, *g*, *b*, *c*, *m*, *y*, *k* in 0—255; *t* in 0—255 (0 is opaque, 255 is fully transparent), negative value may be used to "punch through" text background/highlight colour, with given degree of transparency for foreground colour). 0 is the default value for transparency, i.e. fully opaque. ; Each colour component may be given as a base 256 decimal value (with one decimal). Unless it is negative, the parameter *i* is ignored (and can be omitted, but not the separators), so is the parameter *a*. The transparency part of the foreground colour applies also to inline images/graphics (like emoji), which otherwise keep their colours, in a multiplicative manner.

Of course, for readability, the text foreground colour should be in sufficient contrast to the effective background colour (whether that is via text background/highlight colour or the plane 1 background). There is no automatic contrast guarantee.

- **CSI 38**[**:0**]**m** Reset foreground colour to default foreground colour. The 0 can be omitted (**CSI 38m**). Same as **CSI 39m**. (Default is taken from top of style stack, if style stacking is implemented; and is the inherited setting in case of inheritance.)
- **CSI 38:1m** Fully transparent foreground. Short for **CSI 38:2::0:0:0:255m**. Not recommended, but useful for conversion from Teletext. Deprecated. Note that "punch-through" (of near background, see colour plane exposition above) is given as a negative transparency value here, the more negative, the more "far background", and less "near background", is seen through the text.
- **CSI 38:2:**[*i*]**:***r***:***g***:***b*[**:***t*[**:***a***:0**]]**m** Foreground colour as RGB(T) (the separator here is colon).
- **CSI 38:3:**[*i*]**:***c***:***m***:***y*[**:***t*[**:***a***:0**]]**m** Foreground colour as CMY(T) (the separator here is colon).
- **CSI 38:4:**[*i*]**:***c***:***m***:***y***:***k*[**:***a***:0**]**m** Foreground colour as CMYK (the separator here is colon).
- **CSI 38:5:***p*[**:***t*]**m** Foreground colour from colour palette. Uses the same palette as for background colour. *p* is in an implementation defined range, but not negative. If a transparency value is given, *t*, that overrides the transparency value from the palette.
- **CSI 38:6m** Copy current background colour to foreground colour. Not recommended. Useful (only) for making conversion from ISCII more convenient. Deprecated.
- **CSI 38:7m** Copy current all around shadow colour to foreground colour. Not recommended. Useful (only) for making conversion from ISCII more convenient. Deprecated.
- **CSI 38:8m** Optionally transparent foreground. Like **CSI 38:1m**, but can be "revealed". Useful (only) for making conversion from Teletext more convenient. Deprecated.
- **CSI 38:9:***b***:***p1***:**[*t1*]**:***p2*[**:***t2*]**m** Gradient glyph colouring. *p1* (start colour) and *p2* (end colour) are indices in the colour palette. *b* is an angle between **180=** (−180) and **180** degrees, 0 degrees is up. Negative value is clockwise, positive anti-clockwise. The angle gives the direction of the gradient, which starts and stops just within the "em box". For gradient

purposes, glyphs should be considered to be horizontally centred in the em box. *t1* and *t2*, if present, are transparency overrides.

- **CSI 39m** Reset to default text foreground colour (may be settable in preferences); for subcomponents, resets to the inherited background/highlight colour.

The text foreground colour is inherited to text subcomponents: to table cells, to math expressions (all parts), to vector graphics (text/label parts; perhaps with the condition that the text/label parts are well integrated with ECMA-48 formatting).

8.20.3 Short forms for colours (extended, and colours are given fixed values)

The colours here are a compromise and should be used for new or updated implementations. For a palette, possibly user settable (see section 8.20.7 below), use **CSI 38:5:***n*[:*t*]**m** or **CSI 48:5:***n*[:*t*]**m**. The colours for the short forms are *not* fetched from the palette, they are fixed.

Most terminal emulators nowadays use some variety of dull colours rather than clear colours for shortcut numbers in the 30 and 40 ranges but also have the clear colours at higher numbers after the **CSI** (and, for compatibility, we follow that here). These colours are here fixed (just like the named colours for CSS), to get an (as close as possible) reliable colour output.

x in **3**, **4**: **CSI 3***y*:… foreground; these colours are fully opaque colours (*t*=0)
CSI 4*y*:… background; these may be partially transparent (transparency is implementation defined, may be settable via preferences or **APC** control)

CSI *x***0**[:*t*]**m** is short for **CSI** *x***8:2::0:0:0:***t***m**	Pure black ■ (CSS Black)
CSI *x***1**[:*t*]**m** is short for **CSI** *x***8:2::205:0:0:***t***m**	Dull red 🟥 (Xterm’s Red)
CSI *x***2**[:*t*]**m** is short for **CSI** *x***8:2::0:205:0:***t***m**	Dull green 🟩 (Xterm’s Green)
CSI *x***3**[:*t*]**m** is short for **CSI** *x***8:2::255:215:0:***t***m**	Dull yellow 🟨 (CSS Gold)
CSI *x***4**[:*t*]**m** is short for **CSI** *x***8:2::0:0:205:***t***m**	Dull blue 🟦 (CSS Mediumblue)
CSI *x***5**[:*t*]**m** is short for **CSI** *x***8:2::205:0:205:***t***m**	Dull magenta 🟪 (Xterm’s Magenta)
CSI *x***6**[:*t*]**m** is short for **CSI** *x***8:2::0:205:205:***t***m**	Dull cyan ■ (Xterm’s Cyan, ≈ CSS DarkTurquoise)
CSI *x***7**[:*t*]**m** is short for **CSI** *x***8:2::255:255:255:***t***m**	Pure white (CSS White)

Largely following existing implementations, we define some additional short forms for fixed background and foreground colours. This is a proposed addition, but most terminal emulators already implement some variety of clear/pure colours here.

x in **9**, **10**: **CSI 9***y*:… foreground, reset by **CSI 39m**, *z* is **3**; fully opaque colours (*t*=0)
CSI 10*y*:… background, reset by **CSI 49m**, *z* is **4**; these may be partially transparent (implementation defined)

CSI *x***0**[:*t*]**m** is short for **CSI** *z***8:2::105:105:105:***t***m**	Dark grey ■ (CSS DimGrey)
CSI *x***1**[:*t*]**m** is short for **CSI** *z***8:2::255:0:0:***t***m**	Clear red 🟥 (CSS Red)
CSI *x***2**[:*t*]**m** is short for **CSI** *z***8:2::0:255:0:***t***m**	Clear green 🟩 (CSS Lime)
CSI *x***3**[:*t*]**m** is short for **CSI** *z***8:2::255:255:0:***t***m**	Clear yellow 🟨 (CSS Yellow)
CSI *x***4**[:*t*]**m** is short for **CSI** *z***8:2::0:0:255:***t***m**	Clear blue 🟦 (CSS Blue)
CSI *x***5**[:*t*]**m** is short for **CSI** *z***8:2::255:0:255:***t***m**	Clear magenta 🟪 (CSS Magenta)
CSI *x***6**[:*t*]**m** is short for **CSI** *z***8:2::0:255:255m:***t*	Clear cyan ■ (CSS Cyan)
CSI *x***7**[:*t*]**m** is short for **CSI** *z***8:2::220:220:220:***t***m**	Light grey ⬜ (CSS Gainsboro)
CSI *x***8**[:*t*]**m** is short for **CSI** *z***8:2::169:169:169:***t***m**	Medium grey ■ (CSS DarkGrey)
CSI *x***9**[:*t*]**m** is short for **CSI** *z***8:2::255:140:0:***t***m**	Orange 🟧 (CSS DarkOrange)

8.20.4 Text emphasis mark colour (new, but has implementations)

Underline, overline, CJK emphasis lines/marks. *p*, *r*, *g*, *b*, *c*, *m*, *y*, *k*, *t*, *i*, *a* as above.

- **CSI 58[:0]m** Reset text emphasis mark colour to follow foreground text colour. Default. The 0 can be omitted (**CSI 58m**). (Note: does not reset to inherited text emph. colour.)
- **CSI 58:1m** Fully transparent text emphasis. (Not recommended.) Shadow, if any, will still be visible.
- **CSI 58:2:[*i*]:*r*:*g*:*b*[:*t*[:*a*:0]]m** Text emphasis colour as RGB(T) (the separator here is colon).
- **CSI 58:3:[*i*]:*c*:*m*:*y*[:*t*[:*a*:0]]m** Text emphasis colour as CMY(T) (the separator here is colon).
- **CSI 58:4:[*i*]:*c*:*m*:*y*:*k*[:*a*:0]m** Text emphasis colour as CMYK (the separator here is colon).
- **CSI 58:5:*p*[:*t*]m** Text emphasis colour from colour palette. Uses the same palette as for background colour.
- **CSI 59m** Reset emphasis colour to follow text foreground colour. Reset any colour set by **CSI 58:…m** as well as **CSI 68:…m**. (Note: does not reset to inherited text emph. colour.)

8.20.5 Text overstrike and text framing colour (new)

Overstrike (strike-through), encircling lines, framing lines (not for block frames), margin lines. *p*, *r*, *g*, *b*, *c*, *m*, *y*, *k*, *t*, *i*, *a* as above.

- **CSI 68[:0]m** Reset text crossed-out/strike-through and text framing colour to follow foreground text colour. The 0 can be omitted, **CSI 68m**.
- **CSI 68:1m** Fully transparent overstrike and framing lines. (Not recommended.)
- **CSI 68:2:[*i*]:*r*:*g*:*b*[:*t*[:*a*:0]]m** Text overstrike and text framing colour as RGB(T) (the separator here is colon).
- **CSI 68:3:[*i*]:*c*:*m*:*y*[:*t*[:*a*:0]]m** Text overstrike and text framing colour as CMY(T) (the separator here is colon).
- **CSI 68:4:[*i*]:*c*:*m*:*y*:*k*[:*a*:0]m** Text overstrike and text framing colour as CMYK (the separator here is colon). (Fully opaque.)
- **CSI 68:5:*p*[:*t*]m** Text overstrike and text framing colour from colour palette. Uses the same palette as for background colour.

8.20.6 Text shadow colours (new)

Outline and shadow are included as styles in ISCII, and shadow is supported (with lots of variation) via CSS. The shadow styling can be used to produce (attached-to-glyph) shadow, outline and bevel effects. Text shadows are also useful (and is used when not using text background) for such things as subtitles (or other text overlaying an image/video), that should be readable for a large variety of background colours (in a video image).

Note that the full glyph casts a shadow, that can be seen through transparent glyphs. Glyphs can have several shadows, usually of different colours. How overlapping shadows display is implementation defined. Also transparent, including fully transparent, glyphs cast shadows. The shadows here *need not* behave exactly like “real” shadows, despite the name; rather they behave more like CSS shadows for text.

e is an angle between **180=** and **180** degrees, 0 degrees is “light” (creating the shadow) from top of “paper”, negative is clockwise, positive is counter-clockwise movement of the “light”.
d in **1** (0.01em shadow), **2** (0,02em), ..., **99** (0,99em shadow).
f in **0** (no blur, default), **1** (0,01em blur), ..., **99** (0,99em blur), the blur makes the shadow

gradually more transparent, the distance here is from the outer edge of the shadow.
r, *g*, *b*, *c*, *m*, *y*, *k*, *t* as above.

Note that the parameter *d* and the parameter *f* differ in interpretation here compared to how these parameter positions(!) are interpreted in other colour control sequences.

- **CSI 80**[**:0**]**m** Cancel all-around shadow, directional shadow and its “counter-shadow”. All-around shadow is on colour plane 3.5. The all-around shadow does *not* itself cast a shadow. (Note: does not reset to inherited all-around shadow.)
- **CSI 80:2:**_d_**:**_r_**:**_g_**:**_b_[**:**_t_[**:**_f_]]**m** All-around shadow colour as RGB(T).
- **CSI 80:3:**_d_**:**_c_**:**_m_**:**_y_[**:**_t_[**:**_f_]]**m** All-around shadow colour as CMY(T).
- **CSI 80:4:**_d_**:**_c_**:**_m_**:**_y_**:**_k_[**:**_f_]**m** All-around shadow colour as CMYK. Fully opaque.
- **CSI 80:5:**_d_**:**_p_[**:**[_t_][**:**_f_]]**m** All-around shadow colour from palette. *p* is palette index.
- **CSI 80:6:**_d_[**:**_f_]**m** Copy current foreground colour to all-around shadow colour. Not recommended. Useful for making conversion from ISCII more convenient.
- **CSI 80:9:**_d_**:**_b_**:**_p1_**:**[_t1_]**:**_p2_[**:**[_t2_][**:**_f_]]**m** Gradient all-around glyph shadow colouring. *p1* (start colour) and *p2* (end colour) are indices in the colour palette. *b* is an angle between **180=** and **180** degrees, 0 degrees is towards top of “paper”. The angle gives the direction of the gradient, which starts and stops just within the “em box” of each (base) character. For gradient purposes, glyphs should be considered to be horizontally centred in the em box.
- **CSI 81**[**:0**]**m** Cancel directed shadow, and its “counter-shadow” (if any).
- **CSI 81:**_e_**:2:**_d_**:**_r_**:**_g_**:**_b_[**:**_t_[**:**_f_]]**m** Directional shadow colour as RGB(T).
- **CSI 81:**_e_**:3:**_d_**:**_c_**:**_m_**:**_y_[**:**_t_[**:**_f_]]**m** Directional shadow colour as CMY(T).
- **CSI 81:**_e_**:4:**_d_**:**_c_**:**_m_**:**_y_**:**_k_[**:**_f_]**m** Directional shadow colour as CMYK.
- **CSI 81:**_e_**:5:**_d_**:**_p_[**:**[_t_][**:**_f_]]**m** Directional shadow colour from palette.
- **CSI 81:**_e_**:6:**_d_[**:**_f_]**m** Copy current foreground colour to directed shadow colour. Not recommended. Useful for making conversion from ISCII more convenient.
- **CSI 81:**_e_**:9:**_d_**:**_b_**:**_p1_**:**[_t1_]**:**_p2_[**:**[_t2_][**:**_f_]]**m** Gradient directional glyph shadow colouring. *p1* (start colour) and *p2* (end colour) are indices in the colour palette. *b* is an angle between **180=** and **180** degrees, 0 degrees is up. The angle gives the direction of the gradient, which is just within the em box of each (base) character. For gradient purposes, glyphs should be considered to be horizontally “centred” in the em box.
- **CSI 82:…m** “Counter-shadow”. Same parameters as for **CSI 81:**_e_**:…m**, except that there is no *d* parameter, the inherited direction is 180 degrees opposite to the (just set) direction for **CSI 81:**_e_**:**_d_**:…m**. Using a “counter-shadow” can be used to get a bevel-like or embossed effect.

Note: If using **CSI 80:...m** and the foreground colour is same as the (fill or text) background opaque colour, that can give an outline effect. Multiple shadows, and with the foreground colour same as the (text or fill) background opaque colour, that can give an embossed effect, depending on colour selection for the shadows. Compare the use of shadows in HTML/CSS, as illustrated in: https://codepen.io/daryl/pen/yAuGj, (note `background-clip:text`); https://www.midwinter-dg.com/permalink-7-great-css-based-text-effects-using-the-text-shadow-property_2011-03-03.html.

The text shadow colour(s) are inherited to text subcomponents: to table cells, to math expressions (all parts), to vector graphics (text/label parts; perhaps with the condition that the text/label parts are well integrated with ECMA-48 formatting).

8.20.7 Colour palette (extended with capability to be set via control sequences)

The colour palette, accessible via **CSI *s*8:5:*n*[:*t*]m** (*s* in 3, 4, 5, 6) as well as gradient colouring as well as UI text colours, holds a set of colours, and a colour is accessible by an index number (*n*). These colours may be settable via preferences or installation of a colour palette. Another possibility is to have **CSI *s*8:*u*:*n*=:*d*:*e*:*f*:*g*[:*a*:0]m** (*s* in 3, 4, 5, 6 but is ignored, *u* in 2 (RGBT), 3 (CMYT), 4 (CMYK)) set the colour palette at index *n* to the colour given by the rest of the arguments, *without changing the current colour setting*, only the palette. An implementation may limit the portion of the palette that is settable. Note: changes to the palette are *not* reset by **CSI 0m**. If the colour palette at index *n* is changed during the use of that palette index, the effect is implementation defined. It may take effect immediately or take effect when that palette index is next explicitly used (or anywhere there-between). The palette colour settings do not affect the fixed colours available via the short forms for predefined colours. (The Linux console uses an **OSC** command for this kind of setting of the palette colours.)

E.g. https://www.fossmint.com/nord-modern-design-color-theme-palette-for-your-terminal/ has 16 pastel colours that could be set as the colours of index 0 to 15 of the palette (if the implementation allows such setting); this does *not* change the colours of the "named" colours in ECMA-48 which we have fixed here. https://linuxconfig.org/the-best-linux-terminal-color-schemes-for-2019 has a list of various colour palettes, suitable for (a part of) the palette.

At indices 16 to 255 we recommend that the conventional colours should be stored, and be protected from change (though that is still implementation defined).

8.20.8 Negative image text (clarified)

The negative image applies to colour planes 1 to 6, just where the affected text is displayed, and should apply to inline images/graphics (like emoji) which should also be turned into negatives (as in photo negative). Often **CSI 7m** has erroneously been interpreted as switching background and foreground colours. That would not work well when the background/highlight colour is fully transparent (as recommended by default), nor for planes other than 2 and 5; and, importantly, that is in general unlikely to produce a negative image in any sense of "negative".

- **CSI 7**[:*f*]**m** Negative image (if *f* is **0?0**). *f*, default value **0?0**, value between 0 and 1 inclusive. *f* is a factor to use on each RGB colour value *c*; *f* being 0.0 specifies fully negative colour, *f* being 1,0 specifies fully positive colour; final colour *c′*, for each of R, G, B (*c*) is computed as $c' = f\cdot c + (1-f)\cdot(255-c) = 255 - c - 255\cdot f + 2\cdot f\cdot c$, if 8 bits per colour per pixel. Transparency is not affected.
  CMYK values are first converted to CMY by including the *K* value to each of the CMY values, in this way: $C' = C\cdot(255-K)/255$, $M' = M\cdot(255-K)/255$, $Y' = Y\cdot(255-K)/255$, if 8 bits per colour per pixel. Then the CMY values are converted to RGB, and then use the factor.

- **CSI 27**[:*f*]**m** Positive image (if *f* is **1?0**). Same as **CSI 7**[:*f*]**m**, but the default for *f* is **1?0**.

8.20.9 Concealed (censored) text (extended with variants)

"Hide" characters. Not recommended. The concealed text may still be visible if the text is "selected" (for editing, e.g., copying by a copy-paste operation; depending on how "selected" is visibly marked), and a subsequent paste will also reveal the characters. Thus, one should not use "concealed" (nor any other form of transparent) for passwords or other text that is meant not to be extractable. This was not a concern for original ECMA-48, but it is in a modern setting. Indeed, it is common not to show characters in a password at all (Unix approach), or hide the password by displaying substitute characters, like bullets (web page approach).

- **CSI 8**[**:***v*]**m** <u>Concealed/censored text</u>. Text spans are at planes 3 to 6 temporarily fully transparent. This overrides any set or changed colours for the text. Variants: **:0** cancel concealed characters, **:1** (default) fully transparent as described, **:2** instead blur or **:3** pixelate the text to a degree that the text is unreadable, **:4** replace with an Åg-height line (using text foreground colour at the starting point; still cut and paste can reveal the text).
- **CSI 28**[**:***v*]**m** <u>Cancel "concealed text" effect</u>. Same as **CSI 8**[**:***v*]**m**, but the default is **:0**.

The text concealment/censoring is inherited to text subcomponents: to table cells, to math expressions (all parts), to vector graphics (text/label parts; perhaps with the condition that the text/label parts are well integrated with ECMA-48 formatting).

8.20.10 Blinking text (extended with variants)

Cyclically varying transparency (between the set transparency and 255 (fully transparent)) for text spans and at colour planes 3 to 6. If this is used in a context where inline images are allowed (e.g. emoji), the images in the span blink as well. The same for shadows, underlines, text block frames.

- **CSI 5**[**:***f*[**:***v*[**:***p*[**:***w*[**:***t*]]]]]**m** <u>Slowly blinking</u> (less than approximately 2,5 Hz, implementation defined). The default for *f* is approximately **1** to **2** (Hz), exact value is implementation defined. Cancels rapid blinking. Parameters:
  f is approximate frequency in Hz. If **0**, then effectively *v* (inverse special velocity) is **0** and *p* (phase shift) is **0**, regardless of given values (thus no extra transparency if *f* is **0**).
  v is inverse (approximate) velocity in the character progression direction, 10/*v* em/s, where the em is that of the default font size. The default is **0**, indicating "infinite" spatial speed. *v* can be negative (apparent movement opposite the character progression direction). So *v*=**5** gives the speed 2 em/s in the character progression direction. The granularity of this "movement" may be coarse, like per en rather than per pixel.
  p is the approximate phase shift in the unit 45° for the frequency *f*. *p* can be negative, default is **0**, indicating a "leftmost" *minimal* point of extra transparency for the waveform.
  w is the waveform indicator: **:1** sinusoidal, **:2** triangular, **:3** trapezoidal (implementation defined), **:4** rectangular (default). The transparency change need not be very smooth.
  t is timeout (since start of display of that text portion) in seconds. **0** (default) or negative: no timeout. When timeout occurs, the control effectively turns into a **CSI 8m** (for the blinking text span).
- **CSI 6**[**:***f*[**:***v*[**:***p*[**:***w*[**:***t*]]]]]**m** <u>Rapidly blinking</u> (more than approximately 2,5 Hz, implementation defined). Cancels slow blinking. Same variants as for **CSI 5**[**:***f*[**:***v*[**:***p*[**:***w*[**:***t*]]]]]**m**, but the default for *f* is approximately **3** to **4** (Hz), exact value is implementation defined.
- **CSI 25**[**:***f*[**:***v*[**:***p*[**:***w*[**:***t*]]]]]**m** <u>Steady</u> (not blinking). Cancels rapid/slow blinking. Same variants as for **CSI 5**[**:***f*[**:***v*[**:***p*[**:***w*[**:***t*]]]]]**m**, but the default for *f* is **0** (Hz).

The text blinking is inherited to text subcomponents: to table cells, to math expressions (all parts), to vector graphics, to images.

Note that while blinking text, *directly* via HTML, is deprecated in HTML, for HTML there are also other ways to create a blinking or similar "movement" effect. For instance: CSS animation, EcmaScript code altering the transparency, and animated GIFs.

8.20.11 Fadability control (new)

Teletext allows a page to be made transparent, except parts specially marked. This is intended to allow important parts of a page be shown even if the overall page is not shown, instead showing the background image (colour plane 0 or 0.5), in the case of Teletext, the TV image. The intended use is for "flash news" and subtitles (the latter is still a common use for Teletext) to be shown when otherwise watching the TV program. In the Teletext version this allows the rest of the page to fade (completely), except the subtitles (or news flash) which do not fade. (Note that a Teletext page otherwise covers the entire TV image.) (Note that subtitles also need synch data on the sender side (and in a recording), but that is out of scope for this proposal.) This fade is orthogonal to (but composable with) the fade done by blinking.

- **CSI 112:*f*m** <u>Fade control factor</u>. *f* can be fractional (with **?** as decimal marker), value between 0 and 1 inclusive. Factor to use on transparency value (after mapping ECMA-48 value to the range [0, 1], (255−*t*)/255; 0,0 is fully transparent, 1,0 is fully opaque) when set by preference to use this factor (*p*=1). Default is not to use this factor (*p*=0). Final transparency factor is computed as 1−*p*·(1−*f*), where *p* = 1 (from the preference setting) means to use the fade control factor, *p* = 0 means don't use the fade control factor (default); values between 0 and 1 can be used also for the preference setting (this allows for intermediate fades). In a TV-like setting, and *p* = 1, using factor *f* = 0 will show the TV background (colour plane 0), factor *f* = 1 will show the text. Affects also text background (all backgrounds, planes 1 to 3.5) and text decorations; i.e. planes 1 to 6. The nominal computed total transparency, in [0, 1] with 0 as transparent, is multiplied with (1−*p*·(1−*f*)) giving a final transparency (again with 0 as transparent) for planes 1 to 6.

The text fade factor is inherited to text subcomponents: to tables, to math expressions (all parts), to vector graphics (all parts; perhaps with the condition that the text/label parts are well integrated with ECMA-48 formatting).

8.21 MARGIN LINE (new)

A line in the margin for a displayed line of text may be used as a kind of emphasis marking. It is also popular as change marks ("change bars"). Another use for this kind of line is for block quotes (marking one or more 'paragraphs' (as defined here) in entirety, for quotes usually used in conjunction with indenting the lines).

- **CSI 78**[**:**[*v*][**:***p*]]**m** <u>Line in the margin before the beginning-of-line position</u>. Positioned just outside ("before") the minimum of **CSI 69:…m** and **CSI 70:…m** setting. **CSI 78:0m** terminates. Displayed for lines where (non-zero) **CSI 78:*v*m** is in effect, at about 1 en to 1 em before the beginning of lines. Same variants (*v*) as **CSI 4m**, default is **:8**, bold solid line. If there are several different non-zero **CSI 78:…m** in effect for one display text line, the displayed effect is implementation defined. *p* is an index in the colour palette (see above) for the colour of the margin line. Default is current text foreground colour (even if that colour is not in the palette). If started in a table cell, the margin line is inside that cell.
- **CSI 79**[**:**[*v*][**:***p*]]**m** <u>Line in the margin after end-of-line position</u>. Positioned just outside ("after") the **CSI 71:…m** setting. **CSI 79:0m** terminates. Displayed for lines where (non-zero) **CSI 79:*v*m** is in effect, at about 1 en to 1 em after the end of lines. Same variants (*v*)

as **CSI 4m**, default is **:8**, bold solid line. If there are several different non-zero **CSI 79:…m** in effect for one physical text line, the displayed effect is implementation defined. *p* is an index in the colour palette (see below) for the colour of the margin line. Default is current text foreground colour. If started in a table cell, the margin line is inside the cell (at the current nesting level, see PTX below), rather than in the "page" margin.

Note that the margin lines are displayed only in the margin of lines (after line wrapping) where the margin line styling is in effect; it is *not* just considering explicit line breaks, but also automatic ones. Text margin line is *not* inherited to subcomponents.

8.22 FRAMED PARAGRAPHS BLOCK (new)

Framing a block allows for paragraphs (and row cells) to be framed and get a different background colour (colour plane(s) 1.5, see above). These control sequences should occur "between" paragraphs (or row cells), if they occur at other locations the effect is implementation defined. The margin positions of the lines are just outside the *default* start position and *default* end position of lines (line indent control sequences do not affect the framed block line positions); if it is a row cell (see PTX below) that is framed, the lines are drawn at the cell boundaries. Compare framed "div"s in HTML (not framed "span"s). If multiple framed block lines are co-located (from adjacent blocks), the displayed result is implementation defined.

- **CSI 114**[**:**[*v*][**:**[*p1*][**:**[*b*][:[*p2*][:*c*]]]]]**m** Start frame of paragraphs. Same variants (*v*) as for **CSI 4m**, default is **3**, medium solid line, *p1* is the palette (see below) index for the frame colour (default: current text foreground colour), *p2* is the palette index for the frame fill colour, default is transparent (colour plane(s) 1.5). *b* a value in the range 0—15, default 15, read as a bit pattern indicating which sides get the line;
  bit 0 (lsb): line progression direction start side,
  bit 1: character progression start side,
  bit 2: character progression terminating side,
  bit 3 (msb): line progression terminating side.
  Sides not listed inherit line setting from immediately preceding **CSI 114…m** if any (there may be up to four in a row (to allow for different lines on different sides), with different values for *v* and *b*; only the last value for *p2* and *c* take effect), otherwise no line. Terminated by **CSI 114…m** which also starts another block frame *unless* it is *directly* after another **CSI 114…m**, and by **CSI 115m**, and by end of cell. If pagination is supported, and a framed block crosses a page break (automatic or explicit), there is no block line drawn at the page break. *c*=0 allow automatic page breaks within the block (should still avoid typographic widows/orphans), *c*=1 (default) avoid automatic page breaks within the block.
- **CSI 115m** End frame of paragraphs. If the block framing was started in a row cell (see PTX below), the block frame closes automatically at end of that cell. There may thus be multiple levels of block frames, one "outer", and one per row level (there may be rows within cells; see PTX below). Through this mechanism, cells (from PTX rows) may be framed and coloured (the entire cell, not just the text portion, if started at the beginning of the cell (just after a **CSI 1\** or a **CSI 2\**) and auto-terminated at cell end, at a **CSI 2\** or a **CSI 0\**, ideally joining lines of layoutwise adjacent cells, also if in different rows).

While framed paragraph block is *not* inherited to subcomponents, the frame fill colour (colour plane(s) 1.5) is visible also for subcomponents, especially if their background/highlight colour

(colour plane 2) is transparent (which is the normal case). Adjacent table cells with different block lines in the junction get the "heaviest" of the lines displayed.

8.23 "Western to CJK" (new)

In CJK texts, "western" (or European) characters are often set as if they were CJK characters, when in small portions inside of otherwise CJK text, like a name or a single word. (For "half width" forms, only applicable to some Katakana and Hangul letters, we recommend directly using the characters available in Unicode.) For longer "western quotes", this style should not be used.

- **CSI 117m** For Latin, Greek, Cyrillic, ASCII digits and for common (i.e., not script specific) punctuation and symbols, use CJK layout. This gives the listed characters em width (glyph centred in the em width) without changing font. This also changes line break property AL for these characters to ID (other line break properties are unchanged). Other scripts are not affected by this styling.
- **CSI 118m** Normal layout (and line breaking) for the listed character types.

This style is *not* inherited to subcomponents and it is also terminated at bidi B as well as at end of subcomponent (like a table cell or short text in math expression).

9 CHARACTER TABULATION SET ABSOLUTE (2) – HTSA2 (resurrected, improved, replacing HTS, CTC5, TBC3)

CHARACTER TABULATION SET ABSOLUTE (HTSA) has the control sequence **CSI** *t1***;***t2***;**…**;***tn* **SP N** (note the space (**SP**) before the capital **N**, it is part of the control sequence), where each *ti* is a position in the line (from the beginning position of the line, not counting current line indents, assuming a fixed width font, and each position indication thus in the unit *en*). HTSA was deprecated (for unknown reasons) in the fourth edition of ECMA-48. It does have a flaw, which we will fix here.

But the other way of setting tab stops in ECMA-48 (HTS) has a *much* more serious flaw: one must move (by printing something, likely spaces) to the positions where one wants to set the tab stops. Besides outputting something (which is often an absolute no-no for setting tab stops), the positioning is also unreliable, and depend on the advance widths of glyphs (or spaces), which vary in a "proportional" font and between fonts.

9.1 HTSA2 (renewed)

HTSA sets the tab stops without anything getting printed, and the positions are not dependent on individual glyph's advance widths. Here we suggest that HTS (CHARACTER TABULATION SET) and CTC (CURSOR TABULATION CONTROL, which in addition has a misleading name, it is not a cursor control at all) are not to be used in favour of resurrecting HTSA as HTSA2; using **CSI**…**!N** instead of **CSI**…**SP N**. VTS, for setting line tab stops, is in practice already unused.

HTSA does have a flaw (or two). The unit was set by SSU, and we have already proposed not to use other control sequences that depended on SSU as well as not using SSU itself. So, in the resurrected version (HTSA2), we ignore any SSU setting, and have units associated with each of the tab position values. The unit codes (0 to 8) are the same as for SSU (**0** is 'en' of the *current* font, *current* size, not counting **CSI 77:….m** setting). In addition, we allow fractional values, using **?** as decimal marker.

Independent of HTSA2, there are default tab stops every 4 en to 8 en of the *default* font size (implementation defined, may be part of preferences, changing the ("global" to the document) default font size preference may also change the position of the default tab stops).

Each HTSA2 replaces all previously set HTSA2 tab stops (no accumulation). **CSI !N**, i.e. an empty list of positions, in effect resets to using the default tab stops (but keeping the **CSI 70:***u***:***v***m** tab stops). Until the last, in the character progression direction, explicitly set tab stop (that is HTSA2 or **CSI 70:***u***:***v***m** tab stop), the HTSA2 or **CSI 70:***u***:***v***m** tab stops overshadow the default tab stops.

- **CSI** [*u1*:]*p1*;*…*;[*un*:]*pn***!N** <u>Set character tabulation tab stops according to the stoplist.</u> *n* may be 0 (empty list). All tab stops set by a previous HTSA2 are removed (not those set by **CSI 70:….m**). The stop list is a semicolon-separated unordered list of positions from the default start of line in the character progression direction. Positions are given as [*unitcode***:**]*value*, where the unit code is from the SSU unit codes, and the value is a (possibly fractional) value indicating the distance (when combined with the unit) from *default* start of line in the character progression direction (so tab stops do not move when changing paragraph line indents). The unit can be omitted, and the default unit code is then **0** (en).

The values (with unit) are *each* an offset from the default *beginning* of line (in the character progression direction), not from the previous tab stop (hence "absolute"). E.g., **CSI 5;11?5;15!N** sets tab stops at 5 en, 11,5 en and 15 en from the default beginning position of lines (in the character progression direction; *not* considering **CSI 70:….m** setting). When en is used as unit for a tab stop, it needs to be associated with the current absolute value of en or internally convert the position value that uses en as unit to use a non-en unit. Note that while already set tab stops must not be changed even if the size of en changes (by a font size change), follow-on HTSA2 control sequences are affected by a change of en, if the en unit is used.

The tab positions need not be ordered in the HTSA2 control sequence. However, it is recommendable that the tab stop positions should not be closer to each other than 1 em (2 en). The tab positions may of course be sorted internally, to make their use (internally) easier.

Default tab stops that are 1 em or more *after*, in the character progression direction, the *furthest* explicitly set by HTSA2 (or **CSI 70:….m**) tab stop are still in effect. Default tab stops before that are overridden by the HTSA2 tab stops. The HTSA2 control sequence should occur only at beginning of a line. At other positions, the result is implementation defined.

HTSA2 tab stops are *not* inherited to any subcomponents, like table cells, and tab stops cannot be set in subcomponents, except for tab stops set by line indent setting (**CSI 70:….m**) which are settable in table cells.

10 PARALLEL TEXTS – PTX (clarified and extended)

PTX, as specified in ECMA-48 5th ed., have two completely different uses. While both are "parallel texts" in some sense, one is most easily conceived of as a table row, original use-case was for having the "same" text in multiple languages "in parallel", and the other is for so-called Ruby annotations as used in Japanese and Chinese (though for Chinese there is a fallback to use a parenthetical instead). Unfortunately, they start with the same control sequence in ECMA-48 5th ed. Note: Unlike most other things in ECMA-48, PTX has a structure that can be nested (tables in table cells, and Ruby text in table cells, though tables in Ruby text and Ruby in Ruby are unlikely to be useful). Note that bidi controls (ECMA-48 or Unicode) as well as style settings are autoterminated at cell end, as the cells are bidi and style processed each in isolation.

10.1 Multiple texts (cells) in a single (table) row (extended)

This defines a control sequences for a single table row with (formatted) text content in each cell of the row. Multiple rows with a single NLF (all interpreted as LS) between each row form a table regardless of line spacing and indent, which are *not* applied for the rows. There should be an NLF just before the table, as well as just after. But if an "inline" table is desired, the lead NLF or final NLF or both can be omitted, but then the table should consist of a single cell, which has only(!) the real table in it (nesting), and that table in turn does not need initial nor terminating NLFs, since it is all there is in the single inline cell. This is for *enabling* aligning the table cells to a table.

- **CSI 1**[**;***u***:***s*]\ Beginning of first text/cell (of the list of parallel texts/cells) (i.e., start of first cell in a table row). *u***:***s* gives the initial (minimum) row hight (as unit and measure; in the line progression direction); default is current line advance as set by **CSI 72:***u***:***s*[**:***sb*[**:***sa*]]**m**. The row height can be increased to accommodate the content. Note that **CSI 1\** is also used for Ruby, but then there can be no *u***:***s* part.
- **CSI 2**[**;***u***:***s*[**;***h*[**;***v*]]]\ End of previous text and beginning of next text (i.e., cell boundary). *u***:***s* gives the initial width of the terminated cell (in the character progression direction), default is the width of content without automatic line breaking. The cell width can be increased when aligning cells in adjacent rows into columns of the table.
  h = 0 (default): Use inherited setting for line justification in the character progression direction.
  h = 1: Flush lines so that they begin as set by **CSI 69:***u***:***v*[**:***s*]**m** / **CSI 70:***u***:***v*[**:***s*]**m** within the cell.
  h = 2: Flush lines (the part after beginning of line, or if tab stops are used then after last **HT**/**HTJ** in the line; preceding part is treated as for *h* = 1) so that they end as set by **CSI 71:***u***:***v*[**:***j*]**m** (space/tab characters, at end of the line are not counted) within the cell.
  h = 3: Centre lines between the positions set by **CSI 70:***u***:***v*[**:***s*]**m** and **CSI 71:***u***:***v*[**:***j*]**m** (spaces/tabs at end of line are not counted) within the cell; if there is any HT/HTJ in the line then the centring is done between "its" tab stop and position set by **CSI 71:***u***:***v*[**:***j*]**m** (preceding part is treated as for *h* = 1). (Note that **CSI 69:***u***:***v*[**:***s*]**m** setting is not used.)
  h = 4: Stretch lines *that are not at end of paragraph nor ends with (CR)VT/LS/(CR)FF* (otherwise, treated as for *h* = 1) between (the nearest of) BOL (**CSI 69:***u***:***v*[**:***s*]**m** and **CSI 70:***u***:***v*[**:***s*]**m**) or (if any is used) latest *used* tab stop, and EOL (**CSI 71:***u***:***v*[**:***j*]**m**) positions (if possible) *that are not at end of paragraph nor ends with (CR)VT/LS/(CR)FF* (otherwise, flush to BOL) (spaces/tabs at end of line are not counted) to fill up the line space in the cell by widening SPs, NBSPs, IDSPs, though that should be avoided for spaces before/after words in certain cursive scripts (e.g. Arabic) where the words are instead lengthened by adding ARABIC TATWEEL to some of the words.
  h having a negative value, **1=**, **2=**, **3=**, **4=**: Same as for the corresponding positive values, but no automatic line breaking.
  h = 5 or *h* = 6 (numerical alignment): No automatic line breaking. First align lines (in each lines final tab field; all should use the same final tab field) in the table column (*h* = 5 or *h* = 6 cells only) on first occurrence of COMMA, FULLWIDTH COMMA or ARABIC DECIMAL

SEPARATOR (*j* = 5) or to FULL STOP or FULLWIDTH FULL STOP (*j* = 6) (align with end of line if no occurrence of those characters in the line), then align the entire column to end of line as set by **CSI 71:***u***:***v*[**:***j*]**m** (assuming that the line contents at all fits). This is for numeric alignment of (LTR) numerals, one numeral in the last used tab field per line in the column;
h = 7: Declares the cell as a spanned cell (cannot be used for the first row of a table), content is concatenated (with a **PS** between, if both are non-empty) with the corresponding cell in the column (using multiple table rows), if any, in the directly preceding table row. Text block border and text block background colour are ignored, and those of the preceding cell are used. Note that spanning in the character progression direction is handled by the width specifications.
v = 0 (default): Use inherited (*v* = 1, if top level table) setting for line justification in the line progression direction.
v = 1: Flush the text lines in the cell to the start position within the cell as set by **CSI 72**[**:***u***:***s*[**:***sb*[**:***sa*]]]**m**.
v = 2: Flush the text lines in the cell to the end position in the cell as set by **CSI 72**[**:***u***:***s*[**:***sb*[**:***sa*]]]**m**.
v = 3: Centre the group of lines in the cell between “start of text lines” and “end of text lines” within the cell as set by **CSI 72**[**:***u***:***s*[**:***sb*[**:***sa*]]]**m**;
Styling (SGR, SPD(2), HTSA2, SCO(2)) and bidi are processed for each cell in isolation. Thus, all styling is reset to that inherited for each cell, and bidi controls are also autoterminated at end of each cell.

- **CSI 0**[**;***u***:***s*[**;***h*[**;***v*]]]**** End of list of parallel texts (i.e. end of cell and end of table row). *u***:***s* as above. *h* and *v* as above. The row syntax is **CSI 1***first cell*(**CSI 2**[**;***ux***:***sx*[**;***hx*[**;***vx*]]]*****follow-on cell*)***CSI 0**[**;***ux***:***sx*[**;***hx*[**;***vx*]]]****. This corresponds in principle to the HTML: **<tr><td>***first cell***</td>**(**<td>***follow-on cell***</td>**)***</tr>**, but with cell initial sizing (a bit different from HTML) and a different way of doing cell text positioning.

Matrices in math expressions are not a good fit for this kind of tables. See the separate math expression proposal for how to represent matrices in math expressions.

Note that lines drawn around the cells, as well as cell background colour is set via FRAMED BLOCK (see section 8.22). There is no automatic framing, and the default for the cell background colour is transparent. While frameless may be fine for having parallel texts in different languages, tables usually require framing. Nested tables should use multiple cell background colour planes (see note about colour plane 1.5).

The layout is unspecified in ECMA-48, but multi-column display is what is mentioned (“*the strings may* [here interpreted as “shall”, to get a stringent, and generally useful, interpretation of PTX] *be presented in successive lines in* ***parallel columns****, with their beginnings aligned with one another and the shorter of the paragraphs followed by an appropriate amount of "white space".*”). This was intended for translations or similar but, with the extensions described above, is actually more general and can be used for any kind of table row. Note that cell texts may have several instances of the Ruby variant of PTX (see 10.2), and also nested use of this kind of PTX parallel texts (i.e., table rows within a table cell). Tab stops and line indents are inherited and are interpreted from “beginning of cell” for each cell.

Automatic line breaking is done on text within each cell (if given a width), and the cell height (width for vertical lines) is the maximum cell height in the row. The beginning and end of lines should have about 1 en margin, both horizontally and vertically, to the cell border (but modified by SGR control sequences). Multiple contiguous rows (with explicit line break between) form a table, but there is no automatic alignment of cell borders, and columns form only if there is alignment. There are no automatic lines drawn between table cells, but framed blocks (see section 8.22 above) can be used to both frame a cell and give it a background colour.

Note that **CSI 1\**…**CSI 2\**…**CSI 2\**……**CSI 0\** (PTX) is one of the few mechanisms in ECMA-48 that specifies nesting; the others are the bidi controls. Note also that bidi algorithm needs to be applied in isolation for each PTX cell (if bidi is enabled), and that each a PTX row is a single “object” for the bidi algorithm outside of the sequence of PTX rows. I.e., each table row, as a whole, must be atomic for bidi processing. A table row’s cells are laid out according to the current character progression direction. And the line and character progression directions are inherited to each cell, but can be set per cell via SPD. Further, bidi, if enabled, applies to each cell of the row, in isolation.

There must be a line break between each **CSI 0\** (ending a row) and **CSI 1\** (starting a new row) in order to create a table of rows, where the rows are laid out in the line progression direction. A (single) PTX row with no line break before nor after, will be an inline table row; note that a cell in such a row can have a table. Cells in each row should have consistent widths, with the exception of cells that span columns.

If the NLF between rows is (CR)FF, and there is pagination, that NLF causes a page break. There may also be an automatic page break between rows. Whether there can be a page break within a row, is currently implementation defined. A future version of this specification may define controls for this.

10.2 Japanese/Chinese pronunciation annotations of words/phrases (clarified)

This is the other (completely different) use for PTX. It has nothing to do with tables. The principal text and the phonetic annotation here are not expected to contain RTL characters, nor bidi controls.

- **CSI 1\** <u>Beginning of principal text to be phonetically annotated</u>. The principal text should be a “word” or short phrase. This must not contain any PTX control sequences. Note: there must be no parameters to this use of **CSI 1\**.
- **CSI 3\** or **CSI 4\** <u>End of principal text and beginning of Japanese (3) or Chinese (4) phonetic annotation text (Ruby text).</u> The phonetic annotation text is usually Hiragana or Katakana text for Japanese, Pinyin (i.e., Latin) or Bopomofo text for Chinese. There can be only one such phonetic annotation per principal text. The principal and supplementary texts must *not* contain any PTX control sequences. **CSI 3\** does not allow for parenthesis fallback, but **CSI 4\** does (compare **<rp>** in HTML).
- **CSI 5\** <u>End of Chinese/Japanese phonetic annotation</u>. The syntax for this use of PTX is **CSI 1\***principal text***CSI** (**3** or **4**)**\phonetic annotationCSI 5\**. This corresponds in principle to the HTML **<rb>***principal text***</rb><rt>***phonetic annotation***</rt>**. All SGR styling is reset to that at the start of the Ruby text (Ruby text must be parsed out, just like tables, math expressions, end emoji ligatures) but the phonetic annotation is usually rendered slightly smaller than the principal text. That size change is not automatic but must be given as a control sequence.

This corresponds to the Ruby markup in HTML (https://www.w3.org/International/articles/ruby/markup.en) for producing so-called Ruby text, i.e. phonetic annotations written above (horizontal text lines) or to the right (vertical text lines) of the principal text in Japanese or Chinese (though for Chinese, ECMA-48 suggests the alternative of using a parenthetical inline annotation). For Ruby layout details, see text on HTML's Ruby markup. The parentheses fallback is laid out in the character progression direction.

For bidi (if enabled) and styling (including character progression direction), handle each Ruby as a table with two cells.

11 SELECT CHARACTER ORIENTATION (2) – SCO/SCO2

11.1 SCO (clarified)

The SCO – SELECT CHARACTER ORIENTATION control sequence, **CSI** *v* **SP e**, where the unit for *v* is 45°, is not very clearly defined in ECMA-48 5th edition:

> SCO is used to establish the amount of rotation of the graphic characters *[i.e., combining sequences, or grapheme clusters]* following in the data stream. The established value remains in effect until the next occurrence of SCO in the data stream.
>
> […]
>
> Rotation is positive, i.e. counter-clockwise and applies to the normal presentation of the graphic characters along the character path. The centre of rotation of the affected graphic characters is not defined by this Standard.

This seems to refer to rotating *each* combining sequence (or even grapheme cluster) *individually*. That kind of rotation can be seen in some signage (like for shops, e.g., Latin letters upright while the character progression direction (see SPD) is from top to bottom), but that is rare for text documents, even though it may have uses for table cells and vector graphics. Indeed, this use would need further clarification, regarding interaction with SPD, advance width between characters (read: combining sequences or grapheme clusters), and more, in a future version of this proposal.

11.2 SCO2 (new)

We will here make a different interpretation, for rotations that are commonly used in text documents. We will use a different control sequence, SCO2: **CSI** *v***!e**. We will keep the unit as 45°, but allow negative values, preferring values between (inclusive) 90° (**2**) and −90° (**2=**), fractional values may be allowed for table cells.

- **CSI** *v***!e** outside of a table cell. Rotate each page (entire document if no pagination). For page rotation, only allow integer multiples of 90° (**0**, **2, 4**, **2=**). It should occur at the beginning of document (before anything is output) or just after a hard page break, (CR)FF), assuming that there is pagination for the document, otherwise it has no effect. At other positions, the effect is implementation defined. The page rotation does *not* accumulate, i.e., the rotation indication is always relative to the nominal upright. A page rotation setting is in effect for all subsequent pages until another page rotation is set. A dynamic display (i.e., not print) may autorotate the page display so that it is easier to read.

- **CSI** *v***!e** inside of a table cell. Rotate table cell. For rotating the displayed content of a table cell. Automatic line breaking is turned off *unless* the rotation is an integer multiple of 180° (**0**, **4**). When used as a table cell rotation control sequence, it should occur (at most once) only at the very beginning of a table cell and affects only the content of that table cell, otherwise the effect is implementation defined. It is autoreset at end of cell.

The rotation of a cell combines with the *inherited* rotation (if any, and there can be multiple inheritance levels), and this is the rotation that is inherited into subcomponents: table (tables are allowed inside tables), math expressions, vector graphics, …; *each in their entirety*.

12 SELECT PRESENTATION DIRECTIONS (2) – SPD/SPD2

ECMA-48 has a mechanism for setting text presentation direction, SELECT PRESENTATION DIRECTIONS (SPD). SPD selects the line and character progression directions. The default should be interpreted as left to right character progression, and top to bottom text line progression. An application may also have a (global) user preference SPD setting, used unless SPD is implemented and overrides that preference. SPD has the format **CSI** *c*[**;0**] **SP S** (*c* is a code for which directions are set; the second argument here (**0**) shall(!) be omitted). We will here introduce an *augmentation* to SPD, SPD2, **CSI**…**!S**. All of the alternatives for SPD control sequences are *without* any bidi processing, while the additional alternatives for SPD2 control sequences *can* turn on bidi processing, including the setting the paragraph bidi level. While this notionally is a “mode”, it is not formally an ECMA-48 “mode”. Note that all ECMA-48 “modes” are very strongly deprecated, including BI-DIRECTIONAL SUPPORT MODE.

- **CSI** *c* **SP S** or **CSI** *c***!S** outside of a table cell. Set progression directions and Unicode bidi processing on/off for following paragraphs. When used as a page presentation directions control sequence (i.e., outside of any table cell), it should occur at the beginning of the document (before any content to be displayed), or, if there is a notion of pagination and there is a change of line progression direction, just after (before any content to be displayed) a hard page break ((CR)FF). If not at the beginning of document, it should occur just after (before any content to be displayed) a paragraph break (PS, (CR)LF, NEL). At other positions, the effect is implementation defined. A multipage document should not allow for bottom-to-top line progression direction.
- **CSI** *c* **SP S** or **CSI** *c***!S** inside of a table cell. Set progression directions and Unicode bidi processing on/off for following paragraphs *within the cell only*. When used as a table cell presentation directions control sequence, it should occur just after start of the table cell or just after a paragraph break but in the latter case then no change of line progression direction. At other positions, the effect is implementation defined. Note that a hard page break ((CR)FF) inside of a table cell is interpreted as a hard line break ((CR)VT), LS). The presentation direction set in a cell is autoreset at end of cell.

Note that this is *not* a rotation, and there is no accumulation. Furthermore, these directions are notionally applied *before* any SCO2 rotation (of page or cell). The setting is inherited into some subcomponents: table rows, vector graphics (text labels only), partially into math expressions (bidi on/off only, math expressions are always LTR character progression direction).

12.1 Some problems with the bidi algorithm

The automation the the bidi algorithm brings, works reasonable only for simple prose text. In various instances where the text is not very simple prose, it may bring errors and confusion. Indeed, in many cases it only brings havoc, for instance programming languages (where bidi characters are somewhat unlikely or ill-conceived to use), but more importantly in source code for data languages (XML, JSON, CSV, …) where bidi characters are much more likely, especially in a global/internationalisation context. Applying the bidi algorithm ‘as is’ to the source code in a programming language or a data language makes the editor (or IDE) totally useless in the presence of ‘bidi characters’ in the source code; try, and you will see.

A substring of a text that is logically separate from its surroundings (like a quote appearing within quote marks, parenthesised substrings including those that are surrounded by commas, comma separated lists, phrases, either sides of a colon or semicolon, sentences) and the substring or its (near) surrounding contains “bidi characters” (like Arabic, Hebrew and a few other scripts), the logically separate substring may need to be surrounded by bidi isolation control characters, otherwise there is a risk that the bidi algorithm reorders characters to make the reading illogical and wrong. This cannot be built-in to the punctuation characters, leaving it to the “user”/”author” to carefully insert bidi isolation characters in these cases (where bidi does, or may, cause misreading). Unfortunately, there is very little author advice in this regard.

However, for certain situations, in particular source code for programming languages (like C++, Java, Python, …), source code for data languages (XML, CSV (various forms), JSON, …). For these we recommend “tweaking” the bidi algorithm so that bidi processing is applied only to substrings of only letters and substrings of only digits (this may split ‘identifiers’ for bidi), each in isolation (see below). Similarly for math expressions (see separate proposal). This tweak is the way to apply the bidi algorithm in IDEs (*without* using the control sequences below), absolutely not use the bidi algorithm as is.

12.2 Text layout directions and the Unicode bidi algorithm

While SPD specifies text progression directions, we will not enable bidi processing for any of the SPD (**CSI** … **SP S**) directions. Text is laid out strictly in the directions specified, and any rotation of glyphs is given by SCO (**CSI** … **SP e**). These are not “directions” suitable for RTL text, nor for CJK text in vertical text lines. If an SPD is used, processing of combining characters as well as ligatures and kerning is still applicable when specifying directions via SPD.

For SPD2 (**CSI** … **!S**) we will build in application, for some direction specifications, of the Unicode bidi algorithm. This gives a needed parameter, paragraph direction, for the Unicode bidi algorithm when at all enabled. Also some glyph rotations are built-in, in particular for CJK in vertical text lines, when such a direction is specified via SPD2.

Note that many of the steps are applicable for all values of the SPD/SPD2 setting, but explaining where bidi processing (to the extent bidi is enabled) fits in, requires exposing all the steps.

The steps for rendering a piece of text work on a “paragraph” of text, separated by PS-like NLF or bidi B character. Note that a paragraph can *contain* nested structure that in turn may contain paragraphs, such as a table row, which has cells each of which contain paragraphs, math expressions and vector graphics. Thus, the parsing out of table rows and math expressions needs to be done *before* the determination of “paragraphs” separated by PS-like NLF or other bidi B character; note again that PTX structures (table rows) must be parsed out before bidi (as must math expressions), and each PTX cell is bidi processed in isolation, the PTX structure itself appear as a “non-character” (object replacement character) in the embedding text (the same goes for math expressions). Several steps require that the *internal* string representation covers the UCS without character references; so they need to be processed first. This is formulated for the ECMA-48 document format (as updated here), but similar processing must be done for other document format such as HTML and OOXML (Office Open XML, MS Word), which also allow for “paragraphs” (<p>…</p> in HTML), tables, hyperlinks, images, vector graphics (SVG) and math expressions. These steps are not detailed in the Unicode bidi algorithm UTS, since it is so dependent on the exact nature of the document format; so the Unicode bidi algorithm UTS purportedly deal only with “plain text”, leaving more capable document formats to fend for themselves. Applying the bidi algorithm *directly* to the “source” code for any of the more capable document formats (ECMA-48, HTML, OOXML, RTF, ODT, troff, …) is guaranteed to go very very wrong.

Note also that bidi B characters delimit “paragraphs” for bidi. But such characters are (currently) rarely used for data “markup”. Instead we have (commonly): CSV (with various alternatives for ‘,’), JSON, YAML, XML, and a very large variety of custom data formats. For bidi processing, one maybe ideally parse each data format according to its syntactic rules and do bidi processing (also for the “source” code) on individual data items (e.g. for CVS, bidi processing each data cell individually, for JSON and YAML, bidi processing each key and each data value separately). But adapting “paragraph” selection to a large number of data and programming languages is not ideal.

A simpler approach is to apply bidi only to substrings of *solely RTL characters*, each such substring *in isolation*, for data formats as well as source code in programming languages, if bidi is applied at all. This approach does not depend on source code language, and would still leave the source code readable/editable, in contrast to blindly applying the bidi algorithm to source code (data or program) which renders it as gibberish as soon as there are any RTL characters present. Source code is never like prose, which is what the unlimited bidi algorithm is aiming at. Limiting bidi could be set as a preference, but we also define control sequences for limiting bidi.

Here we deal with ECMA-48 formatted “paragraphs” (or other isolated text snippets):

1. Replace each character reference, see section 3.1, with the character it refers to. Note that **ESC [** is a character reference for **CSI** and that **CSI** …_ are also character references. Note also that some character references (which in themselves are not NLFs nor bidi B) may map (reference) to a character which is an NLF or has bidi property B; for instance, **ESC U** maps to NEL, and **CSI**8233_ maps to PARAGRAPH SEPARATOR (both of which are PS-line NLFs). So the given text may need further subdivision into “paragraphs” (once; **ESC ESC** and **ESC CSI** are not valid character references, and thus are (effectively) **SUB**). However, processing **ESC IS***n* is deferred to step 3.
   [First parse out ISn if there is a particular ISn syntax in effect, then tables and Ruby, then math expressions, then SOH/STX/ETX if supported, then bidi B:s that have not been handled]
2. *Before* parsing out bidi ‘paragraphs’ (by PS-like NLF and bidi B delimiters), parse out structures such as table cells (note that tables may nest), Ruby text (tables and Ruby use PTX delimiters in ECMA-48), math expressions (see https://github.com/kent-karlsson/control/blob/main/math-layout-controls-2024.pdf), hyperlinks (see external OSC 8 proposal), as well as vector graphics and images (which may be represented inline as APC

control strings containing hyperlinks or "commands" (SVG, HPGL, …)) in the paragraph, as well as sequences of emojis (and ZWJ), see https://unicode.org/emoji/charts/emoji-list.html. Note that many of the so parsed out subcomponents may have NLFs *inside* of them, so one cannot parse NLFs first and then substructures; it needs to be done in parallel or even to parse out substructures first (table cells, math expressions, …; note the possibility of there being several levels of substructure), *then* split up into paragraphs by paragraph separators (top level and separately in each substructure).

CSI 0m should (implementation defined) imply inserting closing of paragraphs, table rows (**CSI 0\**), and math expressions (multiple **EME** and **ETX**, as appropriate), both of which may nest, including with each other, as well as closing of Ruby texts (**CSI 5\**) and hyperlinks. However, **CSI 0m** is intended only for terminal emulators, and those might not support tables or math expressions; they may support OSC8 hyperlinks, and **CSI 0m** should close such hyperlinks.

Replace each of the parsed-out parts by a, per the paragraph, unique non-character (including the OBJECT REPLACEMENT CHARACTER), indicating which "text object" it replaces. The text objects are stored in an internal table per paragraph.

Some text objects may contain text in turn, like table rows which has cells which have text in them; likewise for Ruby text, hyperlinks, math expressions and vector graphics (with text boxes). Note that a sequence of cells in a table row will be laid out in the character progression direction, regardless of bidi level for the corresponding non-character, just like that the HT/HTJ direction is bidi *level* independent for the HT/HTJ, but sensitive to the *paragraph bidi direction* (set via character progression direction via SPD2).

3. Process **ESC IS1**, **ESC IS2**, **ESC IS3**, **ESC IS4**, deferred from step 1. This will introduce additional "paragraph" breaks (the characters they map to are bidi B, which must be interpreted as paragraph delimiters for bidi "paragraphs"). These exist just in the (discommended) case that one wants to allow **IS***n* characters inside of otherwise **IS***n* delimited data. (We gave some *examples* of using **IS***n* delimited versions of CSV and JSON; just to show that **IS***n* delimiters need not be considered out-of-date.)

4. Convert LRE to LRI (the LRE replacement), RLE to RLI (the RLE replacement), FSI to LRI if the character progression is LR and to RLI if the character progression is RL (i.e., *no* dependence on "first strong bidi" character for FSI, since reading direction needs to be predictable; just as for the paragraph itself), and convert PDF's that close LRE/LRI/RLE/RLI each to PDI.

   Whether the legacy (ECMA-48) bidi control sequences are interpreted is implementation defined. However, this will limit the "range" of the original ECMA-48 bidi controls to the paragraph that contain them, a limitation originally not present. If implemented, **CSI 0]** and **CSI 0[** ECMA-48 bidi control sequences, should be converted to PDF (U+202C), **CSI 1]** should be converted to LRO (U+202D), and **CSI 2]** to RLO (U+202E), **CSI 1[** to LRO (U+202D) if the current direction is RL and to RLO (U+202E) if the current direction is LR. (The LRE and RLE were "kept as is" in the Unicode bidi specification, for backwards compatibility reasons (keep the erroneous/misleading display as is; despite that there are several implemetions that did not follow the bidi specification to the letter). However, ideally, the interpretation should have been changed, to that of LRI and RLI respectively; so we do that here.

5. Transform styling control sequences (including styling push/pop, if implemented) into (within paragraph) non-overlapping style runs, removing the styling control sequences. The styling of a paragraph will initially inherit the style before the paragraph. How a style run is represented is implementation defined but needs to be manageable when substrings are rearranged in the bidi rearrangement step; this rearrangement step may split style runs.

6. Convert any remaining escape sequences, control sequences, control strings, and SCI sequences as in step 2, but the stored data is just the control substring as is. The style range data needs to be adjusted to these substitutions; this step may be done jointly with step 4.
7. Apply steps 3 to 10 recursively as appropriate to the parsed-out parts in the internal per paragraph lookup table. For cells in table rows, it is applied to the cell content, for math expressions only apply them to identifiers and text components, for vector graphics, apply them to text components, for hyperlinks apply it to the display string part of the hyperlink (not the link part).
8. If Unicode bidi processing is enabled, per SPD2 setting, apply the Unicode bidi algorithm *level calculation* to the resulting string (per SPD2 bidi variant), with non-characters for the parsed-out parts (step 2 and step 5), paragraph direction as per SPD2 setting.
9. Compute the width (and height) of the text parts, taking into account the bidi level, glyph width (after stretch, if any) and height (both after SCO rotation (not SCO2), if any), kerning and ligatures, cursive joining, whitespace widths, also taking into account the generated hyphen in case of automatic or semi-automatic (SHY) hyphenation, and of course also the width of “non-characters” (must of course not be removed) standing for tables, math expressions, and also HT/HTJ “widths” (which need be recalculated during automatic line/page breaking) and compute automatic line break positions. Space widening or kashida insertion applies only to the last HT/HTJ separated portion of the text line. Automatic line and page breaking will (effectively or actually) insert LS and (CR)FF at the line break/page break points, preventing character reordering to pass over these line breaks.
10. If bidi processing is enabled: a) Treat HTJ like HT for bidi; treat (CR)VT and (CR)FF like LS. b) If there is any change of styling in a paragraph of text (not counting subcomponents), then split the styling runs at tab, line, page, as well as bidi level breaks. c) Apply the Unicode bidi algorithm’s *reordering part while keeping track of the moved style runs*. Recall that bidi reordering *does not reorder* over tabs, line or page breaks (hard or automatic).
11. The portions of the text line (separated by line breaks and HT/HTJ), as well as HT and HTJ are laid out in the character progression direction, styling as per styling run data, which has been adjusted according to bidi reordering. Table cells in a table row are also laid out in the paragraph progression direction. Math expressions are insensitive to paragraph direction setting (and to all other SPD/SPD2 settings), except for bidi enabled/not enabled, but sensitive to SCO2 rotation (if implemented). When laying out the text line, the non-characters are replaced by the display of the text objects (bidi and style processed as above, each separately) that were parsed out in the initial steps. It is implementation defined what is done with control sequences/strings, etc. that are not interpreted (as bidi controls, styling controls, table rows, Ruby, math expressions, vector graphics, hyperlinks, …); they may each (effectively) be replaced by SUBSTITUTE or REPLACEMENT CHARACTER; they should not be invisible in display.

The Unicode bidi algorithm is intended for “prose” text, text that is supposed to be in natural or constructed “natural-like” languages, where there may be portions that are in an RTL script. That is for anything between tweets and novels or Wikipedia pages. Applying the bidi algorithm (steps 7 and 9) “wholesale” on text that is computer source code of some kind, however, is *not* recommended, even if it may contain RTL text portions, as the source code then might not only be hard to read, but even misleading about what the code actually is, like appear to have swapped arguments to operators which the actual execution of the code will not do, or have a misleading display order for data elements in a source data file (XML, comma separated, …).

Therefore, **CSI 0!S** (no bidi processing) is strongly recommended for all kinds of computer source code, whether program or data. If one wants to have source code where elements such as identifiers, string literals, and comments are bidi processed, then an IDE should be used. An IDE editor can do programming language (or data language) parsing and apply bidi in isolation on each such element.

Further, bidi processing is not appropriate for text that is displayed in vertical lines of text, like for Mongolian or for CJK (though the latter is often presented in horizontal lines in modern texts). Bidi processing is therefore turned off for SPD2 directions that have vertical text lines.

12.3 SPD2 (new; with bidi settings and implicit character glyph rotation)

The (new) SPD2 can set the following text display directions, rincluding bidi paragraph direction. Note again that HT/HTJ, the text portions separated by HT/HTJ, as well as cells (each as a whole) in table rows are always laid out in the character progression direction.

- **CSI 0!S** Character progression left-to-right, text line progression top-down, **no** bidi processing and *no* automatic cursive joining or ligatures for right-to-left scripts. Useful for most scripts including CJK/Hangul in horizontal text lines. If any Indic script is supported, then support the glyph positioning given by https://unicode.org/Public/UNIDATA/IndicPositionalCategory.txt. If the Mongolian script is supported, glyphs for Mongolian characters are rotated 90° clockwise (in some fonts they may already be so rotated) and are *not* cursively joined. This covers also so-called "visual order" (and pre-joinable glyphs for Arabic) for right-to-left scripts. Ligatures and kerning (though not for fixed-width fonts) is supported. This, and the similar **CSI 0 SP S**, are suitable for display/editing of program source code (C++, Java, bash scripts, etc., etc.) and source data (CSV, XML, etc., as well as HTML source data); SPD2 settings that allow for bidi rearrangement (below) are ***not*** suitable for displaying source code or source data of any kind, since the very structure of the source code/data will get messed up display. If some bidi processing is desired for source code/data, then one need to use special purpose display that apply bidi in a very restricted manner, such as each identifier individually; see **CSI 88!S**. Note that when bidi processing is not enabled, bidi controls are displayed as SUB/REPLACEMENT CHARACTER.
- **CSI 1**[**:***a*]**!S** CJK/Hangul vertical, text line progression right to left, **no** bidi processing. See https://www.unicode.org/Public/UNIDATA/VerticalOrientation.txt for rotation information. Characters that are non-CJK, non-Hangul, non-Mongolian, non-wide, but are strong left-to-right or are symbols get their glyphs rotated 90° clockwise, strong right-to-left characters get their glyphs rotated 90° counter-clockwise. Variant *a*: **0** do not rotate arrow/arrowlike symbols (arrows refer to external directions; autoreset at bidi S and bidi B characters), **1** (default) rotate arrow/arrowlike symbols 90° clockwise (arrows refer to the text progression directions; as given in VerticalOrientation.txt).
- **CSI 2**[**:***a*]**!S** Mongolian vertical, text line progression left to right, **no** bidi processing. Characters that are non-CJK, non-Hangul, non-Mongolian, non-wide, but are strong left-to-right or are symbols get their glyphs rotated 90° clockwise, strong right-to-left characters get their glyphs rotated 90° counter-clockwise (compare VerticalOrientation.txt, but the rotation for Mongolian vertical is *counter*-clockwise), in addition mirrored Ps/Pe characters are mirrored before the rotation. Cursive joining is done for Mongolian characters. Variant *a*: **0** do not mirror nor rotate arrow/arrowlike symbols (autoreset at bidi S and bidi B characters), **1** (default) arrow/arrowlike symbols are mirrored and then rotated 90° counter-clockwise.
- **CSI 3**[**:***a*]**!S** Character progression right-to-left, text line progression top-down, *with* Unicode bidi processing. This setting sets the bidi "*paragraph direction*" to RL. Mongolian

characters are considered bidi strong RTL and their glyphs are rotated 90° clockwise. Cursive joining (only in RTL runs), and ligatures, are done for right-to-left scripts that have that feature (and the script are supported by the implementation), and direction not overridden by LRO. This progression setting is suitable for text that is principally written in an LTR script, but may contain occasional words or phrases that are written in an RTL script, but the text is still **not** program source code/source data. Glyph mirroring, for RTL portions of the text, is normally fine for brackets of various kinds, but for other symbols glyph mirroring or lack thereof may be problematic. Note that for math expressions (properly bracketed by SME…EME), where automatic bidi mirroring is prohibited, symbols that do not have a mirror character can be mirrored by MMS, MATH MIRROR SYMBOL, control code. Bidi processing may easily thwart the semantics of a text, especially for program source code and data source code, but may on occasion, e.g., in-sentence enumerations or simple math expressions that are not protected by **SME**…**EME**, also be an issue for "prose text" (and then recommend using bidi controls to make the text less easy to misread). Variant *a*: **0** do not mirror arrows/arrowlike symbols (Unicode bidi algorithm default; arrows refer to external directions; autoreset at bidi S and bidi B characters), **1** mirror also arrows/arrowlike symbols in reversed runs (default here; arrows refer to the text progression directions as per bidi).

- **CSI 8**[**:***a*]**!S** Character progression left-to-right, text line progression top-down, *with* Unicode bidi processing. This setting sets the bidi "*paragraph direction*" to LR. Mongolian characters are considered bidi strong RTL and their glyphs are rotated 90° clockwise. Cursive joining (only in RTL runs), and ligatures, are done for "bidi" scripts that have that feature (and the script are supported by the implementation), and direction not overridden by LRO. This progression setting is suitable for text that is principally written in an RTL script. Glyph mirroring, for RTL portions of the text, is normally fine for brackets of various kinds, but for other symbols glyph mirroring or lack thereof may be problematic. Note that for math expressions (properly bracketed by SME…EME), where automatic bidi mirroring is prohibited, symbols that do not have a mirror character can be mirrored by MMS, MATH MIRROR SYMBOL, control code. Bidi processing may easily thwart the semantics of a text, especially for program source code and data source code, but may on occasion, e.g., in-sentence enumerations or simple math expressions that are not protected by **SME**…**EME**, also be an issue for "prose text" (and then recommend using bidi controls to make the text less easy to misread). Variant *a*: **0** do not mirror arrows (Unicode bidi algorithm default; arrows refer to external directions; autoreset at explicit (not by automatic line break) bidi S and bidi B characters), **1** mirror also arrows/arrowlike symbols in reversed runs (default; arrows refer to the text progression directions as per bidi).
- **CSI 10!S** Within a paragraph, every odd line is left-to-right, every even line is right-to-left, text line progression top-down. Boustrophedon. No bidi processing, no cursive joining. For some ancient texts, where all glyphs are mirrored when going right-to-left.
- **CSI 11!S** Like **CSI 10!S** but odd lines are right-to-left. Extremely rare use, except possibly for mirroring words like "AMBULANCE"…
- **CSI 13**[**:***a*]**!S** KISS bidi, RTL. Like **CSI 3!S**, but bidi (and cursive joining) is applied using simplified bidi character properties: AL → R, EN,AN → L, ES,ET,CS → ON, and (as for **CSI 3!S**) the "embedding" to "isolate" and mapping of FSI as above.
  a) The Unicode bidi algorithm has some "tweaks" mostly for numbers, making the result of bidi less predictable as well as sensitive to things it should not be sensitive to. This option eliminates those tweaks. b) Bidi mirroring is done only on general category Ps/Pe characters, plus arrows as set by the variant *a*. Variant *a*=**0**: do not mirror arrows (arrows refer to external directions; autoreset at bidi S and bidi B characters); *a*=**1**: in addition to

mirror pair Ps/Pe characters mirror also arrows/arrowlike symbols (but not other symbols) in reversed runs (default; arrows refer to the text progression directions as per bidi).

- **CSI 18**[**:***a*]**!S** KISS bidi, LTR. Like **CSI 13**[**:***a*]**!S** but the bidi paragraph direction is LTR and character progression is left-to-right.
- **CSI 30**[**:***a*]**!S** KISS2 bidi, RTL. Like **CSI 13!S**, but *nestable* (not **RLM**/**LRM**) bidi controls are not interpreted (display as SUB), and bidi S characters are treated as bidi B characters. This is for improved readability of the text; the character order is not affected by bidi controls.
- **CSI 80**[**:***a*]**!S** KISS2 bidi, LTR. Like **CSI 30**[**:***a*]**!S** but the paragraph direction is LTR and character progression is left-to-right.
- **CSI 88!S** Bidi applied to individual "words" only. Like **CSI 80!S**, but bidi (and cursive joining) is applied, *individually*, only to runs of characters that are either only decimal digits all of the same RTL script (rare) or all strong RTL characters that are not decimal digits mixed with combining characters (not first in the run) where all the characters are of the same script ('words'). In either case there must be no strong bidi character before the run, nor after the run (so if bidi R and (non-decimal-digit) bidi L characters are adjacent, bidi will not be applied there).

  This is suitable for source code (C++, Haskell, Java, PHP, …) and source data (CSV, XML, HTML, …), since it does not change the order of "words" (identifiers), they are always displayed left-to-right. Bidi processing otherwise, even with the limitation given for **CSI 80**[**:***a*]**!S**, can often cause severe sematic misreadings, like argument swap and symbol mirroring, which will confuse a reader of the displayed source code or source data, and is impossible to edit.

  This setting is suitable for terminal emulators (often used by programmers) and text editors (e.g. in IDEs) that are used for source code or source data, as well as display of source code/data in, say, textbooks (if they contain source code/data portions containing substrings in a bidi script).

  This approach is similar to the approach for math expressions in *A true plain text format for math expressions (and its XML compatible equivalent format)*, where bidi also must be very strictly limited. Note that there is no glyph mirroring, since symbols are never subject to bidi processing in this directional setting, nor is there any apparent change of argument order (when calling methods, functions, operators, data structure compositions) or data element order, or even worse text jumbling, which is a major issue for less restricted bidi.
- **CSI 99!S** Reset to default directions (and bidi setting), per preference or default.
- **CSI 1=**[**:***a*]**!S** Current progression directions (1, 2, 3, 8, 13, 18, 30, 80; no effect for others except that **99** is a meta-setting). Only change handling of arrow/arrowlike characters as per above. Autoreset to default (**:1**) handling of arrow/arrowlike characters at (explicit) bidi S and bidi B characters. *a* = **1** (default) arrows refer to the text progression directions. *a* = **0** arrows refer to external directions. Change of handling of arrows can occur inside of a paragraph, while all other SPD(2) setting changes apply to at least a whole paragraph (not counting subcomponents).

Through bidi processing, a portion of a paragraph's text can be displayed in the character progression direction "opposite" to that set by SPD2. Note that a paragraph's "bidi direction" is *never* derived from first "strong bidi" character in the paragraph, it is set from SPD2 (or its default). *There must be no future addition of functionality, not even implementation defined, that allows the paragraph direction to be derived from the first "strong" bidi character in the* *paragraph*. The START DIRECTED STRING/START REVERSED STRING control sequences are not to be used in favour to the Unicode bidi control characters.

For automatic line/page breaking, LS (or (CR)VT) or (CR)FF (for page breaking) shall automatically be inserted *before* bidi reversal step (but after bidi level calculation). Note that the bidi reversal step does *not* reorder over (in-paragraph) line breaks ((CR)VT, (CR)FF, LS; not even if automatically generated) nor over character tabulation (HT, HTJ) characters. Thus, glyph (including ligaturing and cursive shaping) widths must be computed before bidi run reversal step. This is specified in the Unicode bidi algorithm. Note that escape sequences, control sequences and control strings must be interpreted (or encapsulated as a subcomponent, unaffected by bidi, if not interpreted) *before* any bidi processing. When inserting LS (or equivalent) when doing automatic line breaking (including language dependent hyphenation, if implemented, and SHY processing; as well as automatic pagination (inserting (CR)FF)) the tab positions moved to by HT or HTJ must be(re)calculated for the part after the newly inserted line (or page) break.

It is important to note that supra-paragraph structure (tables, vector graphics, math expressions; all of which may contain paragraph separators even though the structure itself is inside a paragraph) must be parsed before any bidi processing is done. Exactly how style ranges are represented is implementation defined; but which characters have which style shall not be altered by bidi processing. An embedded graphics or table shall work as if it is a single bidi neutral character for bidi processing. Likewise for each math expression. Note that math expressions are out of reach for the bidi algorithm (except for identifiers, each in isolation) as well as out of reach for any case mapping.

If line progression direction is changed, it is per page: The control should be at the beginning of text or hard page break ((CR)FF (if there is a notion of pagination). If line progression is not changed, but character progression within lines is changed, the change is per "paragraph". The change code should be at the beginning of the part of text affected. If not, the change will take effect only after next explicit break (FF or automatic page break for line progression changes, hard line break except VT/LS/FF for character only progression changes)

An SPD(2) in a table cell only affects that table cell.

13 PAGE FORMAT SELECTION

A replacement for PAGE FORMAT SELECTION may be proposed in a future version of this suggested update, for setting page size, page margins, and page columns (and gutter), and maybe other page related settings.

14 STYLE BRACKETING – Pushing and popping SGR, HTSA2 and colour palette states (new, experimental)

This is an experimental suggestion. It is modelled after an XTerm feature, and may be applicable only to terminal emulator implementations, not so much for text document formatting implementations. This feature should not be used within any subcomponent, like table cells.

In contrast to, e.g., HTML, there is no general nesting, or bracketing, structure in ECMA-48. But PTX, PARALLEL TEXTS, used for tables and Ruby texts, does have a structure and nesting ability (**CSI 1\**.....**CSI 0\**; row start…row end). So does the old bidi controls SDS/SRS (START DIRECTED STRING (**CSI 1]**.....**CSI 0]** and **CSI 2]**.....**CSI 0]**; cmp. LRO…PDF and RLO…PDF), START REVERSED STRING (**CSI 1[**.....**CSI 0[**; cmp. LRO/RLO…PDF, LRO or RLO is selected to be opposite the direction at the point of **CSI 1[**), and so do the Unicode bidi controls (except in the KISS2 variety, see above, which do not allow nesting bidi controls). Math expression, specified in a separate proposal (and *not* a proposed update to ECMA-48) also have a nesting structure.

But XTerm has a nonstandard extension for pushing and popping SGR state. That extension uses private-use “final bytes” (final characters), namely “**{**” and “**}**”. If one wants to standardise this functionality, non-private-use final characters need to be used (with intermediary character(s)). Here we suggest the control sequences:

- **CSI** *n***![** push a copy of all currently set rendition attributes and more: SGR, HTSA2, colour palette plus language, charset (can conceivably change for terminal emulators), as well as font palette. Mark the stack entry with the “GUID”, or non-GUID, *n* (decimal digits only).
- **CSI** *n***!]** pop the rendition attributes stack to and including the topmost level marked with the GUID *n*, using the last popped (GUID *n*) stacked attributes as the currently set ones (both SGR, HTSA2, colour palette, and font palette; possibly also language and charset, but that is implementation defines). **No-op** w.r.t. those settings if *n* is not encountered the stack. For output to a terminal emulator, the GUID should be hard to “guess” if not known. For use inside a document, one may use a very simple non-GUID, like “1” or the current save stack depth.
  All PTX, math expressions, and bidi control nesting are closed, even if the GUID is not encountered in the stack.
  Note that **ESC c** (RESET TO INITIAL STATE, for terminal emulators only) resets even more, if implemented.

15 UI TEXT MARKING (new, not to be stored in docs)

Some applications mark, for instance, search matches, or substrings selected for edit operations (the latter usually only one substring at the time) by using a differently coloured background (exactly which colour varies by application, or even preference setting). Some applications also mark suspected spell errors by a special underline (wavy red seems popular), or suspected style issues (of various kinds) by another coloured underline. Such markings are not part of the styling of the text, but are part of the user interface. An application which does these markings (only) on the “display side” need not invoke any control sequences (or HTML/CSS, or other markup) to display these; and indeed should use different colour layers for those than for the regular background/underline. But for an application where a “remote side” (using ECMA-48 control sequences) decides what to mark that way, it is possible to sometimes repurpose the backgrounds and underlines available via SGR. For such applications (like for applications intended to be displaying the text on a terminal) an ability to “access” the colour planes normally used for those UI markings would be better.

So here we propose UI marking control sequences, which are separate from SGR. UI markings should not be stored in documents. These control sequences are intended for applications where the “remote” side “orders” temporary UI markings, such as “selected”/“matching”, spelling or style “remark”, substring “now being processed”, and similar cases. As opposed to SGR background or underline which cannot have multiple underlines or multiple backgrounds (a limitation compared to some other markup systems), a substring can have multiple UI backgrounds (e.g. both “matching” and “selected”), and multiple UI underlines (e.g. both spell error and style warning), and they do not override the backgrounds or underlines that are part of the document that is displayed (though they may interfere visually). When bidi is used, one span may be split into several separately displayed sections (just as for SGR styles).

- **CSI 0!m** End all UI text marks. Stop all UI text markings.
- **CSI 1:*n*!m** Start UI background colour mark to the colour of index *n* in the palette. *n* is here used both as an index to the palette (same palette as used for SGR) and as an identifier for which UI background is referenced. This way multiple markings (of different palette indices) can overlap, though there is no nesting. Colour plane 2.5. Terminated by **CSI 1:*n*!m** with same value for *n* (restarting with the same colour) or **CSI 2:*n*!m** with same value for *n* or by **CSI 0!m**.
- **CSI 2:**n**!m** End UI background colour mark that started by using index *n* in the palette. Other UI background colour marks are not affected.
- **CSI 3:***n*[**:***v*]**!m** Start UI underline having the colour of index *n* in the palette. *n* here is used both as an index to the palette and as an identifier for which UI underline is referenced. This way multiple markings can overlap (e.g. both spell check marking and style check marking; of different palette indices), though there is no nesting. Same variants as **CSI 4m**, but these lines are a bit lower under the characters (close to the descenders). Default is double wavy underline. Multiple UI underlines on the same substring may stack. If the line progression direction is left to right, the “underline” is to the right of the text. If the line progression direction is right to left, the “underline” is to the left of the text. Colour planes 6.5. Terminated by **CSI 3:***n***:**[**:***v*]**!m** with same value for *n* (possibly changing type of underline, but not colour) or **CSI 4:*n*!m** with same value for *n*, and by **CSI 0!m**.
- **CSI 4:*n*!m** End UI underline that started by using by index *n* in the palette. Other UI underlines are not affected.

Note that, like for “normal” styling, bidi rearrangement may split a UI styling in multiple sections. However, if the UI styling is marking a piece of selected (for editing) text, then there is the option to extend the selection so that it is contiguous also in display.

Also, when the text selection (for editing), and possibly associated UI styling, should encompass “logically sensible text portions”. Exactly what that means will not be defined here, but examples can be seen in modern text editors that allow tables, pictures, or math expressions (using other underlying file format representations, like HTML or some XML format), and possibly also affected by nestable bidi controls, when such are allowed/interpreted. However, it does often include such thing as expanding the selection to more that what is “primarily” attempted to be selected (such as a whole table cell, whole table row, whole math expression, rather than just an “unstructured”/”illogical” part. This may or may not apply to other types of UI markings.

16 Cut and paste

Modern systems offer a “cut and paste” functionality, enabling to save a “clip” in a “clipboard” and then insert that “clip” somewhere else, often via another application. The data saved in the clipboard is in a system dependent format, or even multiple formats. But it usually allows for saving text styling attributes when saving a text clip.

However, it is (and will probably continue to be) unlikely that the styling is saved by using ECMA-48 control sequences in the "clipboard" (though the ECMA-48 control sequences can in principle be stored in the "plain text" version of the clip; recipient programs might not be able to interpret ECMA-48 control sequences). If one does not want to lose the styling attributes (when cutting-and-pasting from an application that does interpret ECMA-48 styling attributes to one that does not, but supports other styling methods), then conversion(s) are needed at some point(s), so that what is saved in the clipboard closely follows the style of the original, to the extent reasonably possible. In addition, saving a clip as "plain text" may or may not keep ECMA-48 style control sequences, whether interpreted or not. That depends on the application and other circumstances.

The clipboard, as mentioned, as well as the receiving application need not support the ECMA-48 way of specifying styling, but one can still get a good, or in ideal cases even perfect, styling copied via the clipboard. E.g., HTML with inline CSS, a possible clipboard format, can represent nearly all the styles mentioned above (some styles, like "margin lines", use of "colour palette", excepted). Further, that format might not be able to fully handle all the line break characters listed above, nor HTJ nor tab stop settings, though LF and functionally equivalent characters can be represented by "<p>" or "</p><p>" (unless within a "<pre>") and VT and functionally equivalent characters by "<br/>" in HTML. In any case one needs to pick up formatting in effect from before the beginning of the text to be copied and reset at the end of the text copied.

When pasting to a terminal emulator, only proper plain text (no ECMA-48 control sequences, nor any other ESC sequences) is often preferable (removing them from "plain text" if need be). An exception could be an ECMA-48 enabled terminal window text editor, but then the pasting needs to convert the styles stored in the clipboard to ECMA-48 control sequences.

When copying from a terminal emulator to an external text editor, it may be a good idea to convert LF to VT (in addition to keeping styling). That way auto-numbered lists (common in rich text editors) will not get messed up if pasting into a list item from a terminal emulator.

17 Control/escape/sci sequences and combining characters

A C0/C1/Cf control character, escape sequence or a control sequence must not occur just before a combining character. Since the final character in an escape sequence or a control sequence, as well as an SCI sequence, is a printable (ASCII) character, often a letter, applying normalisation to NFC may combine that final character and the combining character to a precomposed character, thus "ruining" the escape/control/sci sequence, and misapply the combining character to the wrong base character. In addition, for SCI sequences, note that the SCI may be applied to a Unicode character, not just an ASCII character, the SCI must not be applied to a character that has a canonical decomposition (to two or more characters), since canonical decomposition will corrupt the SCI sequence.

18 Conversion examples

While we will not here take up conversion from all kinds of older, but in some parts similar, text styling representations, we will take up (partially, as sketches only) two of them:

- Teletext. Teletext is still in use (2024) and implemented in all (globally) modern TV sets/TV boxes also those for DVB (Digital Video Broadcasting), including its styling (mostly colouring) mechanisms. We will only cover the "basics", and not the newer extensions (more colours, italics, bold, …) mainly due to the complication that those have "out of line" (i.e., only in the Teletext *protocol*) representations.

- ISCII. ISCII includes, in the standard, mechanisms for styling text. These mechanisms have not been covered (hitherto) in the conversions of ISCII to Unicode.

18.1 Converting Teletext styling to ECMA-48 styling

Teletext is still in common use around the world, especially for optional subtitling, though the use of Teletext for news pages has declined or even been abandoned. Teletext allows for certain style settings, mostly to do with colour. There are also control bits in the Teletext protocol for handling bold, italic and underline. The colour codes also select which set of characters to use (either alphanumeric or "mosaic" characters, the latter are Teletext specific symbols used to build up larger (and crude…) graphics). Note that the coding for "alphanumeric" characters have multiple variants for G0/G2, various Latin subsets, Greek, Cyrillic, Arabic and Hebrew. The used character set is selected among several 7-bit charsets, including national variants of ISO/IEC 646, *except* C0 which ISO/IEC 646 (ECMA-6) requires to be the C0 of ECMA-48/ISO/IEC 6429. These are selected by control bits in the Teletext *protocol*. In addition **0x1B** can be used to switch between a primary and a secondary G0+G2 set. Further, the "mosaic" characters (G1) have two variants, a "contiguous" form (default, and selected by **0x19**) and a "separated" form (selected by **0x1A**). From a modern encoding point of view the latter are separate characters. Teletext also allows for dynamically defined ("bitmapped") fonts of unspecified charsets.

The table is a rough sketch only, far from all details have been worked out. For instance, Teletext overrides (italics, bold, underline, more colours, proportional font, G3 characters) are not covered here, though the functionality is covered by this proposed update to ECMA-48. The styling and codepage selection via the Teletext *protocol* is *not* covered here.

Teletext (style and inline code pg changes autoreset at eol)	**ECMA-48** (as extended here)
0x00 Alpha black and sets "alpha mode", G0 [*not NULL*]	**SP**_?_ **CSI 30m**
0x01 Alpha red and sets "alpha mode", G0	**SP**_?_ **CSI 91m**
0x02 Alpha green and sets "alpha mode", G0	**SP**_?_ **CSI 92m**
0x03 Alpha yellow and sets "alpha mode", G0	**SP**_?_ **CSI 93m**
0x04 Alpha blue and sets "alpha mode", G0	**SP**_?_ **CSI 94m**
0x05 Alpha magenta and sets "alpha mode", G0	**SP**_?_ **CSI 95m**
0x06 Alpha cyan and sets "alpha mode", G0	**SP**_?_ **CSI 96m**
0x07 Alpha white and sets "alpha mode", G0; (default)	**SP**_?_ **CSI 37m**
More colours are available via an out-of-line representation in the Teletext *protocol*.	
Bold, italics and proportional font are available via an out-of-line representation in the Teletext *protocol*.	
0x08 Flash (phases (0°, 180°, …), rates (1Hz, 2Hz) and other effects, like apparent flash movement, specified)	**SP**_?_ **CSI 5m**
0x09 Steady (default) (exceptionally a "set-at" control)	**CSI 25m** **SP**_?_ (note order)
0x0A End box (default) (not in subtitle or news-flash text)	**SP**_?_ **CSI 112:0m** (allow 100% fade)
0x0B Start box (subtitles mainly; was also for news-flashes)	**SP**_?_ **CSI 112:1m** (allow 0% fade)
0x0C Normal size (default) (exceptionally: "set-at" control)	**CSI 77:1:1m** **SP**_?_ (note order)
0x0D Double height (note: background & foreground overrides onto next line, which should be empty)	**SP**_?_ **CSI 77:2:1m** (interaction to next line is implementation defined)
0x0E Double width	**SP**_?_ **CSI 77:1:2m**

Teletext (style and inline code pg changes autoreset at eol)	**ECMA-48** (as extended here)
0x0F Double size (note: background & foreground overrides onto next line, which should be empty)	**SP**_?_ **CSI 77:2:2m** (interaction to next line is implementation defined)
0x10 Mosaics black and sets “mosaics mode”, G1	**SP**_?_ **CSI 30m**
0x11 Mosaics red and sets “mosaics mode”, G1	**SP**_?_ **CSI 91m**
0x12 Mosaics green and sets “mosaics mode”, G1	**SP**_?_ **CSI 92m**
0x13 Mosaics yellow and sets “mosaics mode”, G1	**SP**_?_ **CSI 93m**
0x14 Mosaics blue and sets “mosaics mode”, G1	**SP**_?_ **CSI 94m**
0x15 Mosaics magenta and sets “mosaics mode”, G1	**SP**_?_ **CSI 95m**
0x16 Mosaics cyan and sets “mosaics mode”, G1	**SP**_?_ **CSI 96m**
0x17 Mosaics white and sets “mosaics mode”, G1	**SP**_?_ **CSI 37m**
0x18 Conceal (ignored on the currently displayed page until user presses a ‘reveal’ button on the remote control)	**CSI 38:8m SP**_?_ (_not_ **CSI 8m**)
0x19 (“mosaics mode”) Contiguous mosaic graphics (default); note: does *not* unset underlined in “alpha mode”	(converter change) **SP**_?_
0x1A (“mosaics mode”) Separated mosaic graphics; note: does *not* set underlined in “alpha mode”	(converter change) **SP**_?_
0x1B Called “escape”, but is <u>*unrelated*</u> to ESCAPE, its works like a LOCKING SHIFT *x*: switching between code pages.	SP***?*** **(converter change)** (autoreset at end of line)
7-bit code page settings are available via an out-of-line representation in the Teletext *protocol*.	
0x1C Black background (default) (“set-at”)	**CSI 40m SP**_?_
0x1D Set background colour by copying foreground colour	**CSI 48:6m SP**_?_
0x1E Hold mosaics (details of hold/release mosaics is beyond the scope of this paper) (“set-at”)	(**SP**_?_ **:=** _<current mos. char>_) **SP**_?_
0x1F Release mosaics (details of hold/release mosaics is beyond the scope of this paper)	**SP**_?_ (**SP**_?_ **:= SP**)
line break (implicit) (in Teletext all style settings (and locking shift) are auto-reset for each “row”, i.e., text line; there are *no* line break characters in Teletext, instead the Teletext *protocol* has numbered “rows” of text)	**CSI 0m LF** (**SP**_?_ **:= SP**) ***CSI 0m*** *is used only for convenience here; should use individual resetting controls; also converter c.p. reset*

18.2 Converting ISCII styling to ECMA-48 styling

ISCII formally allows for 31 style settings (many of them outline/shadow) by combining several ATR (attribute; the code for ATR in ISCII is **0xEF**) codes (similar to that ECMA-48 stylings can be combined; but not quite same, vide: ATR HLT ATR BLD does not set independent style attributes, but specifies extra-bold, **CSI 1:3m**). Here we give some of the ISCII styles, and their corresponding ECMA-48 styling codes. The table is a rough sketch only. We will not give the combinations here, just the basic ones.

ISCII (all these ATR toggle)	**ECMA-48** (as extended here) [note: no toggling]
ATR BLD (= ATR **0**) (bold)	**CSI 1:600m** or **CSI 1:700m**, **CSI 22m**
ATR ITA (= ATR **1**) (italic)	**CSI 3m**, **CSI 23m**
ATR UL (= ATR **2**) (underline)	**CSI 4m**, **CSI 24m**
ATR EXP (= ATR **3**) ([width] expand)	**CSI 77:1:2m**, **CSI 77:1:1m**
ATR HLT (= ATR **4**) (highlight/bold) (can be combined with “ordinary bold” to get extra bold)	**CSI 1:600m** or **CSI 1:700m**, **CSI 22m** (**CSI 1:800m**)

ISCII (all these ATR toggle)	**ECMA-48** (as extended here) [note: no toggling]
ATR OTL (= ATR **5**) (outline)	**CSI 80:6:5;38:6m** (setting the all-around 0,05 em shadow colour to be the current text colour, and then setting the text colour to be the same as the (opaque) background colour), **CSI 38:7;89m** (if not combined with ATR SHD)
ATR SHD (= ATR **6**) (shadow)	**CSI 81:45:6:10;80:6:5;38:6m** (extra shadow to the 'south-east'), **CSI 38:7;89m** (if not combined with ATR OTL)
ATR TOP (= ATR **7**) (top half of double height glyphs)	**CSI 77:2:1;113:1m**, **CSI 77:1:1;113:0m** (can be combined with ATR[0-6])
ATR LOW (= ATR **8**) (low half of double height glyphs)	**CSI 77:2:1;113:2m**, **CSI 77:1:1;113:0m** (can be combined with ATR[0-6])
ATR DBL (= ATR **9**) (double size)	**CSI 77:2:2m** or maybe **CSI 76:0:2m** (not clear from ISCII), **CSI 77:1:1m** or maybe **CSI 76:0:0?5m** (can be combined with ATR[0-6])
<line break char> (in ISCII all style settings are auto-reset on explicit line break)	**CSI 0m**, *<line break char>*; ***CSI 0m** is used only for convenience here; should use individual resetting controls; also converter c.p. reset*

ATR [@A-K] and ATR [q-v] signify code page change similar to LS2, LS3, but with an automatic ATR @ ("default code page") at end of line. Thus, they signify a converter change when converting to Unicode. The EXT control code in ISCII, **0xF0**, is similar to the SS2 control code.

19 Security aspects

Filtering out ECMA-48 escape sequences, control sequences or control strings, or "undesirable" control characters, or expanding character references, at the "wrong point" of the processing, may expose security sensitive data (such as blocked commands) that may have been "hidden" by ECMA-48 control/escape sequences, etc., earlier on. These issues have been ever-present in ECMA-48, it is nothing new to this update proposal.

Furthermore, control sequences and control strings (OSC/APC/…) may contain commands that are security sensitive. (Note that *this* proposed update does not define any *control strings* (just their general syntax) at all, though there is hint of some (for HP/GL for instance), other update proposals (or de facto implementations) do define some control strings, e.g. in Xterm (font changes, window titles, …) as well as a hyperlink proposal (now commonly implemented).

The same warnings are valid for any other substring(s) removal, substitution or insertion, such as replacing line breaks with SP (or worse, deleting them, which unfortunately does happen), replacing HT with SPs, removing HTML tags or converting HTML character references, as well as adding explicit line breaks, or removing/replacing portions that have encoding errors, do case mapping or spell correction. All of these are common automatic edits. All of these *may* lead to not only undesirable display, but also security problems (converting a benign command into a problematic one), faulty text analysis (like reading 'word^{a}' (footnote reference) as 'worda') and other issues. These are general issues, not specific to ECMA-48 and not specific to this proposed update of ECMA-48. (A proposal for specific ECMA-48 based *control strings* may have specific security issues, since those may enable access to security sensitive commands; but other text can be read as commands, intentionally or unintentionally, consider shell commands (e.g. shell injection), SQL injection and similar issues, like cross-site scripting.)

Note also that hyperlinks (not a subject in this proposal, but there are already other proposals to have hyperlinks in ECMA-48 based documents) pose a security risk, especially hyperlinks that link “outside” of a site or “secure domain”. Measures should be taken to mitigate those risks. This applies also to images, which indirectly may contain hyperlinks, either as text in the image (can be OCR:d) as QR codes (not all QR codes contain hyperlinks, but they are a popular way of giving hyperlinks indirectly). This proposal does not deal with hyperlinks or images, though some hints are given that can be (or are) used to incorporate them in ECMA-48 based documents.

20 Conclusions

Whether to regard ECMA-48 text styling as a higher-level protocol (for text styling) or not is mostly a matter of taste. Technically it is a (text styling) protocol at the text level. The current principal application, in terminal emulators, relies on it being text level, rather than higher level.

ECMA-48 may seem a bit old-fashioned, since the first edition was issued in September 1976. It has seen little use outside of terminals and terminal emulators, where it is still “alive and well”; though there were some efforts to incorporate parts of it in a document format standard. Much of ECMA-48 is concerned with “terminal only” functionality (some technically possible to apply to modern text editors, also those that do not run “under” a terminal emulator, like cursor movement control sequences, but modern editors use other mechanisms). Despite this, the styling mechanisms can still be used in a modern context, also when storing (styled) text in files. Especially if one modernises the capabilities of ECMA-48 styling, which we are proposing in this paper. Those modernisations can be applied both to terminal emulators as well as text files storing text in a “plain text formatting” format. While one could invent from scratch a different “plain text formatting” format, there seems little need for that considering that we already have ECMA-48. For backward compatibility reasons, terminal emulators will need to stay with ECMA-48. The additions proposed in this paper are in “the spirit of” ECMA-48, and much can apply to terminal emulators as well. With the modernisations, ECMA-48 styling is viable as a text formatting protocol for text files, as well as terminal emulators.

In addition to extending ECMA-48 styling in order to modernise it (in a compatible way), we have also added some functionality to support styling that was present in ISCII, and support for styling that is used in current Teletext use around the world (these days, mostly used for user optional subtitling, though sometimes still in use for news and other pages). With these extensions, ISCII text that use ISCII formatting can be converted to Unicode while preserving the formatting via these ECMA-48 extensions. Likewise, Teletext texts can be converted to Unicode preserving the formatting via these ECMA-48 extensions, supporting the effort to support “old computers” (though Teletext is still in use!). For Teletext, one also need to interpret the Teletext protocol itself for bold/italics/underline, proportional font, line breaks (Teletext has *no* line break characters, instead lines have explicit numbers given in the Teletext protocol), page/subpage numbers (which aren’t text, but gives a structure to the set of Teletext pages being transmitted), as well as charset information (Teletext uses its own charset switching mechanism, out of band and not using any escape sequences).

Possible uses for ECMA-48 (including the extensions proposed here):

- Terminal emulators, which are the current major “client” for ECMA-48. Unfortunately, terminal emulators have a very strict fixed size character cell approach, which in no way is demanded by ECMA-48. But many applications using terminal emulators assume this very strict fixed size character cell approach. Therefore, it may be hard for terminal emulators to adopt features such as proportional fonts, font size as a style, proper tables (PTX), math expressions, vector graphics (“plotter drawings”) and several other features. (Side note: Some terminals, 4 decades ago, could display math expressions, slightly crudely, but apparently no terminal emulator can do anything similar today. And we also had “plotter” terminals 4 decades ago, but no terminal emulator can “plot” today. One can instead generate web pages (or images), but no terminal emulator support…)

- Emulation of “old text applications”. This includes various now defunct computer systems that had control codes similar to those of ECMA-48, such as those using PETSCII, ATASCII, EBCDIC (which for a limited set of systems is still in use), ISCII, as well as Teletext (which is *still* in use; for TV broadcasts and TV sets/boxes, for short news pages and for optional subtitles). This is mostly for archival/library and database conversion use, converting from old character representation to a Unicode based representation of the texts, without losing “control” (like styling) part of the data.

- Text editors that support not just “plain text” but also styled text, tables and math expressions and perhaps vector graphics (diagrams, …). Such text editors are still not “full-featured” document editors, but more capable than “plain text” editors. This is the main motivation for this proposed update/extension to ECMA-48.

20.1 Bonus specifications

ECMA-48 in origin was very explicit in giving bidi directions, no automatic inference at all; however the bidi control sequences correspond to RLO and LRO (and PDF) control codes in Unicode. Unicode introduces the Bidi Algorithm, making direction inferences based on the bidi property of characters. This means that one can largely do without bidi controls for simple running text. It is far from perfect, however, and in many contexts one needs to “tame” it somehow, to avoid misreadings as well as outright garbage. On the one hand, there is complexity (for numbers and currency symbols) that are arbitrary and therefore unhelpful. We propose “KISS” (“Keep It Simple …”) variants that avoid this arbitrary confusing complexity. On the other hand, the Unicode bidi algorithm offer no automation, nor advice, in important cases where the reordering (even in the “KISS” variants) causes ordering errors. One could do the brute force approach of enclosing (begin/end) every sentence, every phrase and every quote, parenthetical part (and maybe more) with isolation bidi controls. However, that is often to overdo it, but there is as yet no handy advice on when to enclose a text part with isolation bidi controls (not even given here…). We do give advice on bidi and programming/data language source text, in short:only apply bidi on RTL letter sequences each in isolation.

In the Unicode (and ISO/IEC 10646) the characters in the so-called C0 and C1 areas are threated with quite a bit of neglect. That includes that most C0/C1 characters are not given reasonable Unicode properties, and indeed C0/C1 is effectively regarded as a private use area. That is quite contrary to the Unicode approach of *not* nilly willy change the semantics of (allocated) characters. However, we will take a little advantage of this looseness, by fixing the C0/C1 area. In Appendices B, C, D, E and F we give more appropriate character properties (name, general category, bidi category, line break category) for C0/C1 characters as of here updated ECMA-48.

21 REFERENCES

[BidiMirroring] http://www.unicode.org/Public/UNIDATA/BidiMirroring.txt.

[CSS] *Cascading Style Sheets* – World Wide Web Consortium, https://www.w3.org/Style/CSS.

[C0/C1 stability] *C0/C1 stability*, Kent Karlsson, https://www.unicode.org/L2/L2022/22013r-c0-c1-stability.pdf.

[DEC-printing] *Digital ANSI-Compliant Printing Protocol – Level 2 Programming Reference Manual*, 1994, Digital Equipment Corporation, https://vaxhaven.com/images/f/f7/EK-PPLV2-PM-B01.pdf.

[ECMA-6] *7-bit coded character set*, 1991. https://www.ecma-international.org/wp-content/uploads/ECMA-6_6th_edition_december_1991.pdf. (Also ISO/IEC 646; note that C0&0x7F controls are updated by [ECMA-48].)

[ECMA-48] *Control Functions for Coded Character Sets*, 5th ed., June 1991, https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-048.pdf.

[ExtraMirroring] *Glyph mirroring: ExtraMirroring.txt*, Kent Karlsson, https://www.unicode.org/L2/L2022/22026r-non-bidi-mirroring.pdf.

[HPGL] *Hewlett-Packard Graphics Language/2*, 1996, https://www.hpmuseum.net/document.php?catfile=213.

[HTML] *HTML* – World Wide Web Consortium (W3C), https://www.w3.org/html.

[ISCII] Indian Standard – *Indian script code for information interchange* – ISCII, 1991. http://varamozhi.sourceforge.net/iscii91.pdf, https://ia800908.us.archive.org/19/items/gov.in.is.13194.1991/is.13194.1991.pdf.

[ISO/IEC 646:1991] Information technology — *ISO 7-bit coded character set for information interchange*, https://www.iso.org/standard/4777.html.

[ISO/IEC 6429:1992] Information technology – *Control functions for coded character sets*, 3rd ed. See [ECMA-48].

[ISO/IEC 8613-6:1994] Information technology – *Open Document Architecture (ODA) and Interchange Format: Character content architectures*. Same as [ITU-REC-T.416].

[ISO/IEC 10646] Information technology – *Universal Coded Character Set* (UCS), https://www.iso.org/standard/76835.html. Same coded characters as in [Unicode].

[ITU-REC-T.416] Information technology – *Open Document Architecture (ODA) and interchange format: Character content architectures*, 1993. https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-T.416-199303-I!!PDF-E&type=items.

[Markdown] *Markdown*; https://www.markdownguide.org/, https://commonmark.org/; and several other variants (Wikipedia, PlantUML, Confluence, …).

[Math controls] *A true plain text format for math expressions (and its XML compatible equivalent format)*, 2025, https://github.com/kent-karlsson/control/blob/main/math-layout-controls-2025.pdf, Kent Karlsson. (Also, in an annex, gives arrow mirror pairs; from [ExtraMirroring])

[OpenType] OpenType features, https://learn.microsoft.com/en-us/typography/opentype/spec/featurelist, https://learn.microsoft.com/en-us/typography/opentype/spec/featuretags.

[OSC8] *Hyperlinks (a.k.a. HTML-like anchors) in terminal emulators*. Egmont Koblinger. https://gist.github.com/egmontkob/eb114294efbcd5adb1944c9f3cb5feda. (Despite the title and the use of OSC, this is *not* oriented to terminals per se; just like the ECMA 48 text styling.)

[PlantUML] https://plantuml.com/, a graphic drawing system geared towards certain diagram types.

[RTF] Word 2003: *Rich Text Format (RTF) Specification*, version 1.8, Microsoft, http://www.microsoft.com/downloads/details.aspx?familyid=AC57DE32-17F0-4B46-9E4E-467EF9BC5540&displaylang=en, 2004.

[SVG] *Scalable Vector Graphics* – W3C, https://www.w3.org/Graphics/SVG.

[Teletext] *Enhanced Teletext specification*, ETSI EN 300 706 V1.2.1, 2003, https://www.etsi.org/deliver/etsi_en/300700_300799/300706/01.02.01_60/en_300706v010201p.pdf.

[Teletext in DVB] Digital Video Broadcasting (DVB); *Specification for conveying ITU-R System B Teletext in DVB bitstreams*, ETSI EN 300 472 V1.4.1, 2017, https://www.etsi.org/deliver/etsi_en/300400_300499/300472/01.04.01_60/en_300472v010401p.pdf.

[TeX] Knuth, Donald Ervin, *The TeXbook*, Computers and Typesetting, 1984, Addison-Wesley, ISBN 0-201-13448-9. (Plus later developments, including LaTeX and more.)

[Unicode] *The Unicode Standard*, http://www.unicode.org/versions/latest/, web site https://home.unicode.org/, http://unicode.org/main.html, including all UAX and UTR.

[Vector graphics] https://en.wikipedia.org/wiki/Comparison_of_vector_graphics_editors.

[Wikitext] Mediawiki syntax, markdown as used in Wikipedia, https://en.wikipedia.org/wiki/Help:Wikitext.

[XTerm control seq.] *XTerm Control Sequences*, 2019, https://invisible-island.net/xterm/ctlseqs/ctlseqs.html.

Appendix A – Control sequences summary

SGR

CSI [**0**]**m** Reset rendition. Handy for things like beginning of prompts, where the current state is unknown. This is intended for terminal emulators. Should not be used in ECMA-48 formatted text files.

CSI 1[:*v*]**m** Bold font variant. Default weight 700.

CSI 2[:*v*]**m** Lean font variant. Default weight 300.

CSI 3[:*v*]**m** Italicized or oblique.

CSI 4[:*v*]**m** Singly underlined. Negative variety code raises the underline while shrinking the underlined glyphs.

CSI 5[:*f*[:*v*[:*p*[:*w*[:*t*]]]]]**m** Slowly blinking. Blinking is mostly for such applications as terminal emulators.

CSI 6[:*f*[:*v*[:*p*[:*w*[:*t*]]]]]**m** Rapidly blinking. Blinking affects transparency, not colour in any other way.

CSI 7[:*f*]**m** Negative image.

CSI 8[:*v*]**m** Concealed/censored characters. Note that this is only a display effect, the text is still there.

CSI 9[:*v*]**m** Crossed out (strike-through) characters.

CSI 10m, ..., **CSI 19m** Palette of font sets (… —lean—normal—bold--extra bold— …; italic/upright; “optical” adaptations for compressed/expanded and different sizes, or a multiple-masters font). (Each should have information if it is calligraphic or not. If not calligraphic, there should be information whether they are sans-serif, serif, or fixed-width.) Note: **CSI 26:0m**: default font set, **CSI 50m**: default fixed width font set, **CSI 26:1m**: default serifed proportional font set, **CSI 26:2m**: default sans-serif proportional font set.

CSI 20m Change to a predetermined Fraktur font set. Italic/oblique does not apply.

CSI 21[:*v*]**m** Doubly underlined. Negative variety code raises the underline.

CSI 22[:*v*]**m** Normal weight (neither bold nor lean). Default weight 400.

CSI 23[:*v*]**m** Roman/upright, i.e. not italicized/oblique. Default. Does not alter bold, underlined, etc.

CSI 24[:*v*]**m** Not underlined. Default.

CSI 25[:*f*[:*v*[:*p*[:*w*[:*t*]]]]]**m** Steady (not blinking). Default.

CSI 26[:*v*]**m** Change to a predetermined serif or non-serif proportional (but non-calligraphic) font set.

CSI 27[:*f*]**m** Positive image. Default.

CSI 28[:*v*]**m** Cancel “concealed/censored characters” effect. Default.

CSI 29[:*v*]**m** Not crossed out. Default.

Foreground colour shorts (default transparency is 0, i.e., fully opaque):

CSI 30[:*t*]**m** is short for **CSI 38:2::0:0:0:***t***m**, Pure black foreground.

CSI 31[:*t*]**m** is short for **CSI 38:2::205:0:0:***t***m**, Dull red foreground.

CSI 32[:*t*]**m** is short for **CSI 38:2::0:205:0:***t***m**, Dull green foreground.

CSI 33[:*t*]**m** is short for **CSI 38:2::255:215:0:***t***m**, Dull yellow foreground.

CSI 34[:*t*]**m** is short for **CSI 38:2::0:0:205:***t***m**, Dull blue foreground.

CSI 35[:*t*]**m** is short for **CSI 38:2::205:0:205:***t***m**, Dull magenta foreground.

CSI 36[:*t*]**m** is short for **CSI 38:2::0:205:205:***t***m**, Dull cyan foreground.

CSI 37[:*t*]**m** is short for **CSI 38:2::255:255:255:***t***m**, Pure white foreground.

CSI 38[:**0**]**m** Reset text foreground colour to default foreground colour.

CSI 38:1m Fully transparent foreground. Short for **CSI 38:2::0:0:0:255m**.

CSI 38:2:[*i*]:*r*:*g*:*b*[:*t*[:*a*:**0**]]**m** Foreground colour as RGB(T).

CSI 38:3:[*i*]:*c*:*m*:*y*[:*t*[:*a*:**0**]]**m** Foreground colour as CMY(T).

CSI 38:4:[*i*]:*c*:*m*:*y*:*k*[:*a*:**0**]**m** Foreground colour as CMYK.

However, if *i* is negative (including negative zero, **0=**), there is no change in foreground colour, but there is an assignment to the colour palette (unless the position is write protected).

CSI 38:5:*p*[:*t*]**m** Foreground colour from colour palette.

CSI 38:6m Copy current background colour to foreground colour. Not recommended.

CSI 38:7m Copy current all-around shadow colour to foreground colour. Not recommended.

CSI 38:8m Optionally transparent foreground. (for Teletext.) Deprecated.

CSI 38:9:*b*:*p1*:[*t1*]:*p2*[:*t2*]**m** Gradient glyph colouring.

CSI 39m Reset text foreground colour to default foreground colour.

Background (or highlight) colour shorts (default transparency is implementation defined, and may be settable in preferences):

CSI 40[:*t*]**m** is short for **CSI 48:2::0:0:0:***t***m**, Pure black background.

CSI 41[:*t*]**m** is short for **CSI 48:2::205:0:0:***t***m**, Dull red background.

CSI 42[:*t*]**m** is short for **CSI 48:2::0:205:0:***t***m**, Dull green background.

CSI 43[:*t*]**m** is short for **CSI 48:2::255:215:0:***t***m**, Dull yellow background.

CSI 44[:*t*]**m** is short for **CSI 48:2::0:0:205:***t***m**, Dull blue background.

CSI 45[:*t*]**m** is short for **CSI 48:2::205:0:205:***t***m**, Dull magenta background.

CSI 46[:*t*]**m** is short for **CSI 48:2::0:205:205:***t***m**, Dull cyan background.

CSI 47[:*t*]**m** is short for **CSI 48:2::255:255:255:***t***m**, Pure white background.

CSI 48[**:0**]**m** Reset background/highlight colour to default text background/highlight colour.

CSI 48:1m Fully transparent text background/highlight (default if more than two colour planes).

CSI 48:2:[*i*]:*r*:*g*:*b*[:*t*[:*a*:**0**]]**m** Text background colour as RGB(T).

CSI 48:3:[*i*]:*c*:*m*:*y*[:*t*[:*a*:**0**]]**m** Text background colour as CMY(T).

CSI 48:4:[*i*]:*c*:*m*:*y*:*k*[:*a*:**0**]**m** Text background colour as CMYK.
However, if *i* is negative (including negative zero, **0=**), there is no change in background/highlight colour, but there is an assignment to the colour palette (unless the position is write protected).

CSI 48:5:*p*[:*t*]**m** Text background/highlight colour from colour palette.

CSI 48:6m Copy current foreground colour to background colour. Not recommended. For Teletext.

CSI 49m Reset background/highlight colour to default text background/highlight colour.

CSI 50m Change to a predetermined “fixed” width font set. (Three widths, width 0 en for non-spacing characters, width 1 en for “narrow” characters, 2 en (i.e., 1 em) for “wide” characters.)

CSI 51[:*v*[:*r*]]**m** Framed string.

CSI 52[:*v*[:*r*]]**m** Encircled string.

CSI 53[:*v*]**m** Overlined.

CSI 54[:*v*[:*r*]]**m** Not framed, not encircled.

CSI 55[:*v*[:*w*]]**m** Not overlined. (Note the additional parameters that modify this.)

CSI 56:1[:*v*]**m** Raised to first superscript level and slightly smaller. Cmp. DEC **CSI?4m** (private use).

CSI 56[**:0**]**m** Not raised/lowered and back to the set size (default). Cmp. DEC **CSI?24m** (private use).

CSI 56:1=m Lowered to first subscript level and slightly smaller. Cmp. DEC **CSI?5m** (private use).
For math (inside **SME**…**EME**), use **SCI^** and **SME_**. See math proposal for details.

CSI 57[**:0**]**m** Reset to default w.r.t. ligatures.

CSI 57:1m Only required ligatures.

CSI 57:2m In addition to required ligatures, use also modern ligatures (according to the font).

CSI 57:*v***m**, v = 3 or greater. Reserved for future use.

CSI 58[**:0**]**m** Reset text emphasis mark colour to follow foreground colour (default).

CSI 58:1m Fully transparent text emphasis (not recommended).

CSI 58:2:[*i*]:*r*:*g*:*b*[:*t*[:*a*:**0**]]m Text emphasis colour as RGB(T).

CSI 58:3:[*i*]:*c*:*m*:y[:t[:*a*:**0**]]m Text emphasis colour as CMY(T).

CSI 58:4:[*i*]:*c*:*m*:*y*:*k*[:*a*:**0**]m Text emphasis colour as CMYK.
However, if *i* is negative (including negative zero, **0=**), there is no change in text emphasis colour, but there is an assignment to the colour palette (unless the position is write protected).

CSI 58:5:*p*[:*t*]**m** Text emphasis colour from colour palette.

CSI 59m Reset emphasis colour to follow text foreground colour.

CSI 60[:*v*]**m** CJK underline (on the right side if vertical character progression direction).

CSI 61[:*v*]**m** CJK double underline (on the right side if vertical character progression direction).

CSI 62[:*v*]**m** CJK overline (on the left side if vertical character progression direction).

CSI 63[:*v*]**m** CJK double overline (on the left side if vertical character progression direction).

CSI 64[:*v*]**m** CJK stress marks (dot placed under/over (right/left side if vertical writing)).

CSI 65m Cancel the effect of the renditions established by parameter values 60 to 64.

CSI 66[**:0**]**m** Normal uppercase and lowercase rendering (default). Implies **CSI 67**[**:0**]**m**.

CSI 66:1m Uppercase letters, 0 to 9, and & rendered as small caps.

CSI 66:2m Uppercase letters, 0 to 9, and & rendered as petit caps.

CSI 66:666m Lowercase letters rendered as the petit caps of their uppercase. Language dependent.

CSI 66:999m Lowercase letters rendered as the small caps of their uppercase. Language dependent.

CSI 67[**:0**]**m** Uppercase fixed width digits (default).

CSI 67:1m Lowercase fixed width digits. (Sometimes misleadingly called “old-style” digits.)

CSI 67:2m Uppercase proportional width digits.

CSI 67:3m Lowercase proportional width digits. (Sometimes misleadingly called “old-style” digits.)

CSI 67:99m Uppercase fixed width digits, where the ‘0’ has a middle dot or an internal diagonal stroke.

CSI 68[**:0**]**m** Reset text crossed-out/strike-through and string framing colour to follow foreground colour.

CSI 68:1m Fully transparent overstrike and framing lines. (Not recommended.)

CSI 68:2:[*i*]**:***r***:***g***:***b*[**:***t*[**:***a***:0**]]**m** Text overstrike and string framing colour as RGB(T).

CSI 68:3:[*i*]**:***c***:***m***:***y*[**:***t*[**:***a***:0**]]**m** Text overstrike and string framing colour as CMY(T).

CSI 68:4:[*i*]**:***c***:***m***:***y***:***k*[**:***a***:0**]**m** Text overstrike and string framing colour as CMYK.
However, if *i* is negative (including negative zero, **0=**), there is no change in overstrike/framing colour, but there is an assignment to the colour palette (unless the position is write protected).

CSI 68:5:*p*[**:***t*]**m** Text overstrike and string framing colour from colour palette.

CSI 69:*u***:***v*[**:***s*]**m** First line BOL (beginning of line) indent. Side is given by SPD2 (explicit or default). Effect starts at current paragraph, with FF, VT and LS counted as inside of paragraph. This, and the two following, should be at the beginning of a paragraph.

CSI 70:*u***:***v*[**:***s*]**m** Non-first line BOL line indent (after auto-line-break). Side is given by SPD2.

CSI 71:*u***:***v*[**:***j*]**m** EOL (end of line) line indent. Also sets paragraph justification. Side is given by SPD2.

CSI 72:*u***:***s*[**:***sb*[**:***sa*]]**m** Line spacing, Intra-paragraph, and before/after paragraph *extra* spacing.

CSI 73m Reserved for HTML-like superscript start. In contrast to all other SGR controls, **CSI 73m**…**CSI 75m** nest. Use instead **CSI 56:1m**, which does not nest. For math (inside **SME**…**EME**), use **SCI^**.

CSI 74m Reserved for HTML-like subscript start. In contrast to all other SGR controls, **CSI 74m**…**CSI 75m** nest. Use instead **CSI 56:1=m**, which does not nest. For math (inside **SME**…**EME**), use **SCI_**.

CSI 75m Reserved for HTML-like superscript/subscript end. Use instead **CSI 56:0m**, which does not nest.

CSI 76[**:***u***:***s*]**m** Font size. **CSI 76m** is superscript/subscript size (just size change); explicit reset via **CSI 76:9m**. **CSI 76m** is a handy shortcut for “smaller size horizontal division” in math expressions; see math expression representation proposal [Math controls].

CSI 77:*a***:***b***m** Font magnification.

CSI 78[**:**[*v*][**:***p*]]**m** Line in the margin before the beginning-of-line position. Side determined by SPD2 setting.

CSI 79[**:**[*v*][**:***p*]]**m** Line in the margin after end-of-line position. Side determined by SPD2 setting.

CSI 80[**:0**]**m** Cancel all-around shadow, directional shadow and its “counter-shadow” (if any).

CSI 80:2:*d***:***r***:***g***:***b*[**:***t*[**:***f*]]**m** All-around shadow colour as RGB(T).

CSI 80:3:*d***:***c***:***m***:***y*[**:***t*[**:***f*]]**m** All-around shadow colour as CMY(T).

CSI 80:4:*d***:***c***:***m***:***y***:***k*[**:***f*]**m** All-around shadow colour as CMYK. Fully opaque.

CSI 80:5:*d***:***p*[**:**[*t*][**:***f*]]**m** All-around shadow colour from palette. *p* is palette index.

CSI 80:6:*d*[**:***f*]**m** Copy current foreground colour to all-around shadow colour. Not recommended.

CSI 80:9:*d***:***b***:***p1***:**[*t1*]**:***p2*[**:**[*t2*][**:***f*]]**m** Gradient all-around glyph shadow colouring.

CSI 81[**:0**]**m** Cancel directional shadow and its “counter-shadow” (if any).

CSI 81:*e***:2:***d***:***r***:***g***:***b*[**:***t*[**:***f*]]**m** Directional shadow colour as RGB(T).

CSI 81:*e***:3:***d***:***c***:***m***:***y*[**:***t*[**:***f*]]**m** Directional shadow colour as CMY(T).

CSI 81:*e***:4:***d***:***c***:***m***:***y***:***k*[**:***f*]**m** Directional shadow colour as CMYK.

CSI 81:*e***:5:***d***:***p*[**:***f*]**m** Directional shadow colour from palette.

CSI 81:*e***:6:***d*[**:***f*]**m** Copy current foreground colour to directed shadow colour. Not recommended.

CSI 81:*e***:9:***d***:***b***:***p1***:**[*t1*]**:***p2*[**:**[*t2*][**:***f*]]**m** Gradient directional glyph shadow colouring.

CSI 82:…**m** “Counter-shadow” to the current directional shadow.

CSI 83m, …, **CSI 89m** Reserved.

Foreground colour shorts (default transparency is 0, i.e. fully opaque):

CSI 90[**:***t*]**m** is short for **CSI 38:2::105:105:105:***t***m**, dark grey foreground.

CSI 91[**:***t*]**m** is short for **CSI 38:2::255:0:0:***t***m**, clear red foreground.

CSI 92[**:***t*]**m** is short for **CSI 38:2::0:255:0:***t***m**, clear green foreground.

CSI 93[**:***t*]**m** is short for **CSI 38:2::255:255:0:***t***m**, clear yellow foreground.
CSI 94[:*t*]**m** is short for **CSI 38:2::0:0:255:***t***m**, clear blue foreground.
CSI 95[**:***t*]**m** is short for **CSI 38:2::255:0:255:***t***m**, clear magenta foreground.
CSI 96[**:***t*]**m** is short for **CSI 38:2::0:255:255:***t***m**, clear cyan foreground.
CSI 97[**:***t*]**m** is short for **CSI 38:2::220:220:220:***t***m**, light grey foreground.
CSI 98[**:***t*]**m** is short for **CSI 38:2::169:169:169:***t***m**, medium grey foreground.
CSI 99[**:***t*]**m** is short for **CSI 38:2::255:140:0:***t***m**, orange foreground.

Background (or highlight) colour shorts (default transparency is implementation defined):
CSI 100[**:***t*]**m** is short for **CSI 48:2::105:105:105:***t***m**, dark grey background.
CSI 101[**:***t*]**m** is short for **CSI 48:2::255:0:0:***t***m**, clear red background.
CSI 102[**:***t*]**m** is short for **CSI 48:2::0:255:0:***t***m**, clear green background.
CSI 103[**:***t*]**m** is short for **CSI 48:2::255:255:0:***t***m**, clear yellow background.
CSI 104[**:***t*]**m** is short for **CSI 48:2::0:0:255:***t***m**, clear blue background.
CSI 105[**:***t*]**m** is short for **CSI 48:2::255:0:255:***t***m**, clear magenta background.
CSI 106[**:***t*]**m** is short for **CSI 48:2::0:255:255:***t***m**, clear cyan background.
CSI 107[**:***t*]**m** is short for **CSI 48:2::220:220:220:***t***m**, light grey background.
CSI 108[**:***t*]**m** is short for **CSI 48:2::169:169:169:***t***m**, medium grey background.
CSI 109[**:***t*]**m** is short for **CSI 48:2::255:140:0:***t***m**, orange background.

CSI 110[**:***z*]**m** Advancement modification.
CSI 111[**:***x*]**m** Space advancement modification.
CSI 112:*f***m** Fade control factor.
CSI 113:*v***m** Show horizontally halved glyphs. (For conversion from ISCII and xterm, otherwise deprecated.)
CSI 114[**:**[*v*][**:**[*p1*][**:**[*b*][**:**[*p2*][**:***c*]]]]]**m** Start frame of paragraphs. Should be at beginning of a paragraph.
CSI 115m End frame of paragraphs. Should be at end of a paragraph. There is no nesting of frames.
CSI 116:*v***m** Overstriking grapheme by grapheme, by slashes, arrows or crosses.
CSI 117:*v***m** Western to CJK layout. Letter/digit/symbol width to em width, and letter/digit to l.b. ID.
CSI 118:*v***m** Reset "western to CJK layout" to normal layout.

HTSA2 (original HTSA remains deprecated)

CSI [*u1***:**]*p1***;**…**;**[*un***:**]*pn***!N** Set tab stops according to the stoplist. **CSI !N** clears all HTSA2 set tab stops.

SCO2 (rotation of pages, rotation of table cell; SCO is *not* deprecated)

CSI *v***!e** Rotate pages (until change of rotation) when at explicit beginning of page, or rotate content of table cell when at the beginning of a table cell. The unit for *v* is 45°.

SPD2 (augmenting SPD, which is *not* deprecated)

CSI 0!S Character progression left-to-right, no bidi processing. HT and HTJ as well as cells in a table row are in left-to-right order. Common default. Also used for *visual order* (when a bidi algorithm and shaping (using ARABIC…FORM characters) has been done separately prior to rendering). BOL is at the left side. Suitable for most scripts (LTR, user not able to read RTL scripts). Very suitable for source code (.java, .css, .cpp, .xml, .json, .cvs, and many other kinds of computer source code) texts, *especially* if there is any RTL script content (no bidi processing or cursive shaping *at all*), since it avoids bidi pitfalls that may occur for source code (but less likely for running text, "prose"). Also suitable for command line interpreters.

CSI 1[**:***a*]**!S** CJK/Hangul vertical, line progression right-to-left, no bidi processing but (in contrast to **CSI 1 SP S**) rotates non-CJK/Hangul characters, including parentheses/brackets and other punctuation. Character progression, HT and HTJ as well as cells in a table row are in top-to-bottom order. Table rows are in right-to-left order. BOL is at the top side.

CSI 2[**:***a*]**!S** Mongolian vertical, line progression left-to-right, no bidi processing but (in contrast to **CSI 2 SP S**) rotates non-Mongolian (and non-CJK/Hangul) characters, including parentheses/ brackets/punctuation. Character progression, HT and HTJ as well as cells in a table row are in top-to-bottom order. Table rows are in left-to-right order. BOL is at the top side.

CSI 3[**:***a*]**!S**, **CSI 13**[**:***a*]**!S**, **CSI 30**[**:***a*]**!S** Character reading direction right-to-left, with Unicode bidi processing; actually sets paragraph direction to "1" for the bidi algorithm, but "printing order" still left-to-right after bidi rearrangement and mirroring; but HT and HTJ as well as cells in a table row are in right-to-left order. BOL is at the right side. Suitable for texts in “bidi scripts” (with occasional occurrences of LTR texts, such as numerals, or Latin script phrases). *Do not use* for source code (.java, .json, …), as it is certain to misprint the code.

CSI 8[**:***a*]**S**, **CSI 18**[**:***a*]**!S**, **CSI 80**[**:***a*]**!S** Character reading direction left-to-right, with Unicode bidi processing; actually sets paragraph direction to “0” for the bidi algorithm, but “printing order” still left-to-right after bidi rearrangement and mirroring; HT and HTJ as well as cells in a table row are in left-to-right order. BOL is at the left side. Suitable for (non-code) text in an “LTR script” with occasional “bidi script” phrases. *Unsuitable* for source code (.java, .json, .cpp, .xml, …), since it *may* cause misreading if RLT text occurs (in identifiers, comments, or string literals) in the source code.

CSI 10!S, **CSI 11!S** Boustrophedon. No bidi, but right-to-left text lines get *all* glyphs mirrored. Historical.

CSI 88!S LTR text with bidi processing, but that is limited to individual words in isolation. For source code.

CSI 1=[**:***a*]**!S** Change arrow mirroring/rotation to referencing external, **0**. Default, **1**, is internal reference.

CSI 99!S Reset to default directions, per preference or default setting. Note that if an SPD2 setting is within a table cell, a pop to outer SPD2 setting is implicit at end of cell.

UI text emphasis marking

For temporary text styling for UI reasons, like marking text selection, spell errors, pattern matches, etc. for terminal emulator-oriented programs (other programs are likely to use other mechanisms for UI text marking). These control sequences should not occur in “**.txtf**” text documents, but are for UI.

CSI 0!m End all UI text marks. Stop all UI text markings. Handy for things like beginning of prompts, where the current state of UI text marking is unknown.

CSI 1:*n***!m** Start UI background colour mark to the colour of index *n* in the palette.

CSI 2:*n***!m** End UI background colour mark that started by using index *n* in the palette.

CSI 3:*n*[:*v*]**!m** Start UI underline, variant *v*, having the colour of index *n* in the palette.

CSI 4:*n***!m** End UI underline that started by using by index *n* in the palette.

PTX as table row

CSI 1[**:***u***:***s*]**** Beginning of first text of the list of parallel texts. Beginning of table row or beginning of base text for Ruby annotation (below).

CSI 2[**:***n*[**:***v*]]**** End of previous text (cell) and beginning of next text (cell) (i.e., cell boundary).

CSI 0[**:***n*[**:***v*]]**** End of list of parallel texts (i.e. end of row).

PTX as Ruby text

CSI 1 Beginning of principal text to be phonetically annotated. Beginning of table row (above) or beginning of base text for Ruby annotation. Any **:***u***:***s* is ignored if this is for Ruby text.

CSI 3 or **CSI 4** End of principal text and beginning of Japanese (3) or Chinese (4) phonetic annotation text (Ruby text). There should be only one of these between **CSI 1** and **CSI 5**.

CSI 5 End of Chinese/Japanese phonetic annotation.

Push/Pop styling

CSI ![push a copy of all currently explicitly set rendition attributes.

CSI *n***!]** pop the rendition attributes stack till it has *n* levels. If *n* is omitted, *n* is current stack size minus one. If *n* is **1=** (and push/pop is implemented), this is a “super-reset” to system defaults.

Some ECMA-48 related possibilities **not** detailed in this proposal

SOS charset=<*IANA charset-name*>**ST** Declare character encoding (akin to XML/HTML/CSS), using IANA charset names, non-ISO/IEC 2022. Should occur only at the beginning of an (ECMA-48 formatted) text file. Note: a declaration, not a codepage “switch”. (All kinds of codepage *switching* have been deprecated/excluded in this proposal.) **SOS lang=**<*IANA language tag*>**ST** Language tagging.

APC hpgl:<*HP-GL/2 commands or* `.hgl` *filename*>**ST**: Embed HP-GL/2 graphics (extended with ECMA-48 formatting and math expressions in ‘labels’ (text subcomponents)). Additions: Allow SGR styling for “labels” (text components of the vector graphics) and allow SCI based math expressions in “labels”. **APC svg:**<*SVG markup or* `.svg` *filename*>**ST** Embed SVG graphics.

Appendix B – Giving C0/C1 and more chars some actually useful Unicode properties

As a bonus, in addition to the styling control sequence updates, we here propose updates to the C0/C1 (and a bit more) parts of Unicode. Since C0/C1 (currently, the general category Cc part) is basically (still) a private use area (with ECMA-48 as a very vague default), it is permissible to do quite a bit of updates to that part of ECMA-48 to fit modern usage and requirements regarding C0/C1. Some of these updates have been hinted on in the main text.

Some may despairingly say that C0 and C1 are all a) “just device controls” and b) “we don’t want to deal with that from a character encoding perspective”. Both of which are utterly false.

Counter-a) Analysing C0/C1 characters, we can regard 15 of the C0 characters as data characters, *not* device controls; all of the (in the ECMA-48 update) C1 allocated characters are data characters. Only some, 14 of them, of the characters in C0 can be regarded as device control characters. The rest of C0 code points should be regarded as undefined.

Counter-b) Not wanting to “deal” with C0/C1 characters has lead to the current stance that C0/C1 are all “private use area” (with ECMA-48 as a “default”). This is of course “upside-down world”, and *absolutely untenable* from a standardisation perspective; some of the *most important characters* in all of computing are not “protected” by being standardised and stable, and the allowed “code page swapping” can wreak havoc. For EBCDIC the “control” characters were not changeable, whereas the “printable” characters could be swapped for different markets (languages). The same goes for the ASCII based systems. The “control” characters could/can not be changed, whereas the “printable” ones could and still can be changed. (Note that C0 characters are the same for both EBCDIC and ASCII! Just moved around.) There are a few historical oddball outliers (like PET and ATARI, but they could not even keep the same control characters between versions). So even though there were (and still are) actual programmatic mechanisms for changing “printable” character “set”, nobody has ever implemented a programmatic switching mechanism for “control” characters. And why is that? Because that would be extremely disruptive to all of computing, since one then cannot rely on even LF, CR and HT to stay constant, cannot even rely on that \0 can be used as string terminator, which is very common in many programming languages. This unreliability is also the reason why the C0/C1 “escape code registry” (its website, https://www.itscj-ipsj.jp/, does not seem to be reachable anymore, and that’s all well and good) is a total failure, apart from that most entries are ill-conceived, are in error, and that the registry is unmaintained, nor does it cover the C1 of EBCDIC.

So leaving all of C0/C1 as private use (as now), is not a good idea. We want to 1) guarantee that LF, CR, HT and \0 are stable, at the very least, 2) also keep data separators (unfortunately not used today, which is a pity, trying to revive, since they were actually a good idea, 3) do some revamping of the actual device control characters (popular systems do use them, but we need some adjustments), 4) adopt math syntax per separate proposal, 5) phase out clearly outdated device controls (and allow their reuse as modern device controls), ECMA-48 has not been updated in over three decades, and some things were obsolete even then). In doing the updates hinted, we do take advantage of that the entirely of C0/C1 is *up till now* regarded as ‘private use area’ (which makes the entries in NameAliases.txt invalid), *then* they should be regarded as fixed (except that there are still some ‘private use’ stuff; indeed, two categories of them: DEVICE CONTROL n for device controls, and INFORMATION SEPARATOR/BRACKET n for data syntax.

Here we give an extended summary of more appropriate property assignments for the C0/C1 characters; name, general category, bidi, line break. We give revised character properties for general category, bidi category, and line break category. All characters that are *not properly interpreted* (like not implemented) nominally display as SUB and *must be visible in display*.

c.p.	Name and comment (if char in C0/C1/Cf/Z*x* is *not* interpreted, it *shall not* be invisible)	g.c.	Bidi	LB
0000	*~~NULL~~* (\0) *(**special use** as string terminator or 'not set')*	Cn	B	CB
0001	START OF HEADING	Zp	B	BK
0002	**START OF TEXT** *(reused for embedded text in math. expr.)*	Zp	B	BK
0003	**END OF TEXT** *(reused for embedded text in math. expr.)*	Zp	B	BK
0004	END OF TRANSMISSION	Cd	B	CM
0005	~~ENQUIRY~~ "**DEVICE CONTROL SIX**"	Cd	BN	CM
0006	~~ACKNOWLEDGE~~ "**DEVICE CONTROL SEVEN**"	Cd	BN	CM
0007	**ALERT** BELL (\a) *(TTY: brief audible or visible alert to user)*	Cd	BN	CM
0008	**CANCEL CHARACTER** (\b) *(was BACKSPACE); this since that is the de facto use of U+0008 in cooked mode tty; inline char. deleter; short for **CSI 1=;0X** in a tty text editor context*	Cd	BN	CM
0009	CHARACTER TABULATION (HT, \t)	Cf	S	BA
000A	LINE FEED (LF, BK ('newline'), \n) *(LF & CRLF used as PS)*	Zp	B	LF
000B	LINE TABULATION (VT, \v) *(no longer a "tab"; used as LS)*	Zl	S	LF
000C	FORM FEED (FF, \f) *(like VT, + page break if pagination)*	Zl	S	LF
000D	CARRIAGE RETURN (CR, \r) *(rarely used alone as PS), note: CRLF is a single Zp, CRVT a single Zl, CRFF a single Zl, CR is sometimes a filler, to be removed/inserted (SMS/CBS)*	Zp	B	CR
000E	(*was* ~~LOCKING SHIFT ONE~~ ***never** interpret in Unicode/10646 texts) Permanently reserved as Cn; Always map to SUB*	Cn	S	CM
000F	(*was* ~~LOCKING SHIFT ZERO~~ — " —) *Permanently reserved as Cn; Always map to SUB*	Cn	S	CM
0010	**DEVICE CONTROL ZERO** *(was ~~DATA LINK ESCAPE~~: effectively **eliminated;** the standards ECMA-24/ECMA-37, **are withdrawn**)*	Cd	BN	CM
0011	DEVICE CONTROL ONE	Cd	BN	CM
0012	DEVICE CONTROL TWO	Cd	BN	CM
0013	DEVICE CONTROL THREE	Cd	BN	CM
0014	DEVICE CONTROL FOUR	Cd	BN	CM
0015	**DEVICE CONTROL FIVE** (*was ~~NEGATIVE ACKNOWLEDGE~~*)	Cd	BN	CM
0016	**INFORMATION SEPARATOR ZERO** (*was ~~SYNCHRONOUS IDLE~~ which was a replacement for ~~NULL~~*)	Cf	B	CB
0017	**DEVICE CONTROL EIGHT** (*was ~~END OF TRANSMISSION BLOCK)~~*	Cd	BN	CM
0018	CANCEL [**LINE**] *(inline deleter, had application-defined extent, but here fixed as erase/CANCEL LINE (in tty input)*	Cd	BN	CM
0019	**DEVICE CONTROL NINE** *(was ~~END OF MEDIUM~~)*	Cd	BN	CM
001A	SUBSTITUTE *(the general/original **REPLACEMENT CHARACTER**)*	So	ON	AI
001B	ESCAPE (\e) *(**never** use for code page switching in Unicode/10646)* *(when received from a TTY: often misused for a should-be INTERRUPT, **ESC a**; note that that relies on timing, since e.g. arrow keys send, via the TTY, control sequences starting with **ESC[**, the char. reference for **CSI**)*	Cf	BN	CB
001C	INFORMATION SEPARATOR FOUR (IS4) *(**private-use separators;***	Cf	B	CB
001D	INFORMATION SEPARATOR THREE (IS3) ***cmp. CSV/JSON data formats;***	Cf	B	CB
001E	INFORMATION SEPARATOR TWO (IS2) ***E48&this upd: no standard***	Cf	B	CB
001F	INFORMATION SEPARATOR ONE (IS1) ***hierarchy or format!**)*	Cf	B	CB
007F	*(was ~~DELETE~~, **eliminated** from ECMA-48)* "**DEVICE CONTROL TEN**"	Cd	BN	CM
0080	***START OF MATH EXPRESSION*** (SME) *(see math expr. proposal)*	Cf	S	CM
0081	***END OF MATH EXPRESSION*** (EME) *(see math expr. proposal)*	Cf	S	CM
0082	[LINE] BREAK PERMITTED HERE (BPH) *(original ZERO WIDTH SPACE)*	Cf	BN	ZW
0083	NO [LINE] BREAK HERE (NBH) *(original ~~ZERO WIDTH NO BREAK SPACE~~ (never use ~~ZWNBSP~~) and original WORD JOINER)* *(inhibits any break opportunity that would be here)*	Cf	BN	WJ
0084	***MATH HORIZONTAL DIVISION*** (MHD) *(see math expr. proposal)* *only has effect inside **START OF MATH EXPRESSION** math expr.*	Sm	S	BK
0085	NEXT LINE (NEL) *(like CRLF but essentially only EBCDIC use)*	Zp	B	BK
0086	START OF SELECTED AREA *(for (outdated) block terminals)* *generalize to **INFORMATION LEFT BRACKET ONE***	Cf	S	CM
0087	END OF SELECTED AREA *(for (outdated) block terminals)* *generalize to **INFORMATION RIGHT BRACKET ONE***	Cf	S	CM

c.p.	Name and comment (if char in C0/C1/Cf/Zx is *not* interpreted, it *shall not* be invisible)	g.c.	Bidi	LB
0088	*(was* ~~CHARACTER TABULATION SET~~ *HTS; use HTSA2 instead)*	**Cn**	**BN**	**XX**
0089	CHARACTER TABULATION WITH JUSTIFICATION (HTJ) *(for itemized lists;* para.dir.==num.dir.*: can use for numeric alignment, e.g. by using HTJ just after the integer part of the numerals, with suitable char tab stop setting and line br.)*	**Cf**	**S**	**BA**
008A	*(was* ~~LINE TABULATION SET~~ *VTS; there is no VTSA)*	**Cn**	**BN**	**XX**
008B	PARTIAL LINE FORWARD (PLD) *(don't use for sup/sub;* ***CSI56…****)*	**Cf**	**S**	**CM**
008C	PARTIAL LINE BACKWARD (PLU) *(don't use for sup/sub;* ***CSI56…****)*	**Cf**	**S**	**CM**
008D	REVERSE LINE FEED *(likely intended for handling columns; moving 'back up' and using tab settings (HTS) for getting the column start positions; nroff could make such output)*	**Zl**	**S**	**BK**
008E	*(was* ~~SINGLE SHIFT TWO~~ ***never*** *interpret in Unicode/10646 texts) Permanently reserved as Cn; Always map to SUB*	**Cn**	**BN**	**CM**
008F	*(was* ~~SINGLE SHIFT THREE~~ *– " –) Permanently reserved as Cn; Always map to SUB*	**Cn**	**BN**	**CM**
0090	~~DEVICE CONTROL STRING (DCS)~~ *(was used in xterm and DEC VTx)*	**Cf**	**BN**	**CM**
0091	PRIVATE USE ONE (PU1) *(use as* ***INFORMATION SEPARATOR FIVE****)*	**Cf**	**B**	**CB**
0092	PRIVATE USE TWO (PU2) *(use as* ***INFORMATION SEPARATOR SIX****)*	**Cf**	**B**	**CB**
0093	*(was* ~~SET TRANSMIT STATE *was intended for half-duplex TTYs)*~~	**Cn**	**BN**	**XX**
0094	***BACKSPACE*** *moved here (was CANCEL CHARACTER); U+0008 is the de facto CANCEL CHARACTER for tty (inline character eraser)*	**Cf**	**BN**	**CM**
0095	*(was* ~~MESSAGE WAITING *alert-ind. & request a DSR report)*~~	**Cn**	**BN**	**XX**
0096	START OF GUARDED AREA *(for (outdated) block terminals) generalize to* ***INFORMATION LEFT BRACKET TWO***	**Cf**	**B**	**CB**
0097	END OF GUARDED AREA *(for (outdated) block terminals) generalize to* ***INFORMATION RIGHT BRACKET TWO***	**Cf**	**B**	**CB**
0098	START OF STRING (SOS) *(a control string starter; OSC/...)*	**Cf**	**BN**	**CM**
0099	***MATH MIRROR SYMBOL*** (MMS) *(see math expression proposal) only has effect inside* ***START OF MATH EXPRESSION*** *math expr.*	**Cf**	**BN**	**CM**
009A	**SINGLE CHARACTER INTRODUCER** (SCI) *(use for math controls) only use for creating new "controls" (Cf) with followon ch*	**Cf**	**BN**	**CM**
009B	CONTROL SEQUENCE INTRODUCER (CSI) *(text styling, kbd input)*	**Cf**	**BN**	**CM**
009C	STRING TERMINATOR (ST) *(control string terminator)*	**Cf**	**BN**	**CM**
009D	OPERATING SYSTEM COMMAND (OSC) *(e.g. xterm: window title)*	**Cf**	**BN**	**CM**
009E	~~PRIVACY MESSAGE (PM)~~ *(text to show in a (TTY) status line)*	**Cf**	**BN**	**CM**
009F	APPLICATION PROGRAM COMMAND (APC) *(e.g. HPGL, named font,…)*	**Cf**	**BN**	**CB**
...	...			
00AD	SOFT HYPHEN *("LINE BREAK WITH HYPHENATION PERMITTED HERE")*	**Cf**	**BN**	**BA**
200B	ZERO WIDTH SPACE *(originally [LINE] BREAK PERMITTED HERE)*	**Cf**	**BN**	**ZW**
2060	WORD JOINER *(originally NO [LINE] BREAK HERE)*	**Cf**	**BN**	**WJ**
2028	LINE SEPARATOR (LS) *(rare, in practice use VT)*	**Zl**	**S**	**BK**
2029	PARAGRAPH SEPARATOR (PS) *(rare, in practice use LF/CRLF)*	**Zp**	**B**	**BK**
...	...			
FFFC	OBJECT REPLACEMENT CHARACTER *(extend with non-characters)*	**So**	**ON**	**CB**
FFFD	REPLACEMENT CHARACTER *(originally SUBSTITUTE)*	**So**	**ON**	**AI**
*n*FFFE, *n*FFFF (with *n* in 0—F), FDD0—FDEF	**Non-characters**. *Primary use*: **object replacement characters** (*per paragraph*, see handling described in the main text). For the primary use, as well as other possible uses, it is not helpful to handle them as unassigned (Cn), it is *much* more helpful to handle them as additional *object replacement characters*.	**So**	**ON**	**CB**

\0, \a, \b, \n, \f, \v \r, \t, \e are character references (often misleadingly called "escape sequences") in C/C++/Java/PHP/bash/… usable in source code character literals and string literals. (Note that the Unix/Linux command *stty* uses a completely different system, though it looks similar at first glance.) There are also octal, hexadecimal (\x) and Unicode (\u, \U) character references in many programming languages. HTML also has a similar system for Unicode character references (&…;), as does this proposal (**CSI**…**_**).

SOH/**STX**/**ETX** were intended for a kind of telegram messaging, no longer used as far as the author knows. However, the **STX** and **ETX** controls we suggest being *reused inside* of math expressions (in a proposed new representation for math expressions), for start/end of embedded short texts, like short explanatory notes as well as for hexadecimal (or other > 10 base) numerals.

The “new” general category Cd (technically Cs, but renamed for clarity) is for device control characters. They must be single byte characters in UTF-8, and not tied to data structuring (i.e. not **STX**, **IS***n*, and similar), nor be used for ECMA-48 general functionality (like **ESC**), nor be for code switching (originally), nor be special use (like ex-NULL, \0). In practice one can reuse the code for Cc to use for Cd, since we have completely eliminated Cc as ill-conceived. There are suggestions for how to use the Cd characters for tty input (and in the solitary case of ALERT BELL: for tty output). These characters are all undefined for raw tty input, and have meaning only for “cooked”/”sane” tty input (except ALERT BELL which only has sense for tty output regardless of “raw/cooked”). Device control characters are undefined for display of text (and are thus to be displayed as SUBSTITUTE or equivalently REPLACEMENT CHARACTER, both visible symbols).

Other ex-Cc characters have been assigned a general category, bidi category and line break category suitable according to character semantics (as per available category values), in a few cases according to reuse semantics as updated here. The names are as per ECMA-48 5th edition in most cases, but with a few updates/changes/deletions as appropriate.

Some hitherto unused control codes (except U+0084, which for a very short time was “INDEX”, a duplicate of LINE FEED) are proposed to be used for math expression representation. See details in *A true plain text format for math expressions (and its XML compatible equivalent format)*, 2025, https://github.com/kent-karlsson/control/blob/main/math-layout-controls-2025.pdf. U+0084 (MHD) can be interpreted as LF when outside of a math expression (xterm). This, and a few other math symbols will be handled as layout control characters in math expressions.

Regarding the **IS***n*, see Appendix F. START/END OF GUARDED/SELECTED AREA are part of ECMA-48’s support for forms on a kind of buffering terminals where one could edit locally and then use a “send” button to send the filled-in “form”. We here just set them to bidi B to make the parts bidi isolated (and should reset SGR), while CB for line break. We will generalize these to be INFORMATION BRACKETs, which are like INFORMATION SEPARATORs, but slightly limited to being bracketing. The block terminal semantics is then a specific private use of these brackets.

Uninterpreted or *faulty* control codes(C0/C1/Cf/Zp/Zl/Zs/Cn/So)/sequences/strings shall **not be invisible** but should nominally ***display*** as SUBSTITUTE/REPLACEMENT CHARACTER or be “better” displayed. This “better” include using the “missing glyph” of OpenType, the glyph for “REPLACEMENT CHARACTER” (many fonts have a glyph for that character), using the “hex boxes” that are common in parts of Linux, or other way of indicating the character, like the alternate (but often “greyed out”) glyphs that some applications use in “show invisibles” mode. Note also that some *interpreted* characters have more than one level of interpretation. For example the math special characters have a “source” level interpretation (showing the source for the math expression, compare TeX source code for math expressions) and a “parsed and laid out” interpretation showing the laid out and formatted math expression (if there is not syntax error). Similarly for **IS***n* formatted data, one may show the source, or as a laid out grid simple data table (if CVS-like, and no errors, and under a private (external to ECMA-48 and this update) agreement as to the syntax). CVS data tables (and JSON data) are often parsed and used, without ever being directly displayed.

But there are different “control codes” in EBCDIC, PETSCII, ATASCII, ISCII, and TeleText… They often cannot be mapped *directly* to (singular) control codes in ECMA-48 (even as updated here). For many of them, they can be mapped to control sequences as per ECMA-48 (especially as updated here, since we have added quite a few control sequences for styling covering ISCII and TeleText, and provided hints for mapping tables for those). Many of these “old” control codes can be mapped straight-forwardly to ECMA-48 control sequences. A few control codes in old encodings cannot be mapped to standard ECMA-48 control sequences. For those, one can use private use control sequences; ECMA-48 have private use control sequences, thousands of them; but it is recommendable to avoid the private use control sequences already defined for use in xterm. A very small number of not-covered control codes in EBCDIC are device controls; most device controls (as updated here; Cd) are private use (DEVICE CONTROL n); make your pick, in case of control codes that are still relevant. Alternatively, map defunct control codes directly to SUBSTITUTE/REPLACEMENT CHARACTER. The details would be for interested parties to work out, not to be given here.

Appendix C: C0/C1/nc for UnicodeData.txt

C0 and U+007F:

Note: Cd (alias for Cc) are the device control characters, they are for ttys; and only in C0

```
0000;<reserved>;Cn;0;B;;;;;N;NULL (never used as char);;;;   (\0) the ultimate (and true) non-character
0001;START OF HEADING;Zp;0;B;;;;;N;;;;;
0002;START OF TEXT;Zp;0;B;;;;;N;;;;;
0003;END OF TEXT;Zp;0;B;;;;;N;;;;;
0004;END OF TRANSMISSION;Cd;0;B;;;;;N;;;;;
0005;DEVICE CONTROL SIX;Cd;0;BN;;;;;N;ENQUIRY (defunct);;;;     reuse dev.ctrl. (gen.)
0006;DEVICE CONTROL SEVEN;Cd;0;BN;;;;;N;ACKNOWLEDGE (defunct);;;;     reuse dev.ctrl. (gen.)
0007;ALERT BELL;Cd;0;BN;;;;;N;BELL;;;;     (\a) avoid name collision
0008;CANCEL CHARACTER;Cd;0;BN;;;;;N;BACKSPACE (moved);;;;    (\b) reflect actual use (since decades)!
0009;CHARACTER TABULATION;Cf;0;S;;;;;N;HORIZONTAL TABULATION;;;;     (\t)
000A;LINE FEED;Zp;0;B;;;;;N;NEW LINE (END OF LINE);;;;    (\n)
000B;LINE TABULATION;Zl;0;S;;;;;N;VERTICAL TABULATION;;;;    (\v)
000C;FORM FEED;Zl;0;S;;;;;N;;;;;    (\f)
000D;CARRIAGE RETURN;Zp;0;B;;;;;N;;;;;    (\r) Zp/B iff not immediately followed by LF/VT/FF nor filler
000E;<reserved>;Cn;0;S;;;;;N;LOCKING SHIFT ONE (SHIFT OUT);;;;      permanently
000F;<reserved>;Cn;0;S;;;;;N;LOCKING SHIFT ZERO (SHIFT IN);;;;      permanently
0010;DEVICE CONTROL ZERO;Cd;0;BN;;;;;N;DATA LINK ESCAPE (defunct);;;;     reuse dev.ctrl.
0011;DEVICE CONTROL ONE;Cd;0;BN;;;;;N;;;;;
0012;DEVICE CONTROL TWO;Cd;0;BN;;;;;N;;;;;
0013;DEVICE CONTROL THREE;Cd;0;BN;;;;;N;;;;;
0014;DEVICE CONTROL FOUR;Cd;0;BN;;;;;N;;;;;
0015;DEVICE CONTROL FIVE;Cd;0;BN;;;;;N;NEGATIVE ACKNOWLEDGE (defunct);;;;     reuse dev.ctrl.
0016;INFORMATION SEPARATOR ZERO;Cf;0;B;;;;;N;SYNCHRONOUS IDLE (synch. NULL);;;;    generalized
0017;DEVICE CONTROL EIGHT;Cd;0;BN;;;;;N;END OF TRANSMISSION BLOCK (defunct);;;; reuse dev.ctrl.
0018;CANCEL LINE;Cd;0;BN;;;;;N;CANCEL;;;;     reflect actual use (for terminals in certain popular systems)
0019;DEVICE CONTROL NINE;Cd;0;BN;;;;;N;END OF MEDIUM (defunct);;;;     reuse dev.ctrl. (gen.)
001A;SUBSTITUTE;So;0;ON;FFFD;;;;N;;;;;    canonically map to REPLACEMENT CHARACTER (conversions map back)
001B;ESCAPE;Cf;0;BN;;;;;N;;;;;    (\e) for C1 char refs and a few device controls, not for code page switching
001C;INFORMATION SEPARATOR FOUR;Cf;0;B;;;;;N;FILE SEPARATOR (ASCII semantics);;;;    gen.
001D;INFORMATION SEPARATOR THREE;Cf;0;B;;;;;N;GROUP SEPARATOR (ASCII semantics);;;;    gen.
001E;INFORMATION SEPARATOR TWO;Cf;0;B;;;;;N;RECORD SEPARATOR (ASCII semantics);;;;    gen.
001F;INFORMATION SEPARATOR ONE;Cf;0;B;;;;;N;UNIT SEPARATOR (ASCII semantics);;;;    gen.
007F;DEVICE CONTROL TEN;Cd;0;BN;;;;;N;DELETE (eliminated in 5th ed. ECMA-48);;;;    gen.
```

(`ALERT BELL` is the only Cd character that is sent *to* a 'tty'; all other Cd characters are sent *from* the 'tty'; for the 'generic' Cd characters (`DEVICE CONTROL` x), there is a suggestion in clause 3 of *ecma-48-keyboard-modernisation-2025-08* that is suitable for Unix/Linux tty:s. For the **IS***n* there is original(!) CSV revival use as well as JSON-like use hinted in the main text. Note that unlike US/RS/GS/FS, the ISn has no *predetermined* hierarchy.)

C1:

```
0080;START OF MATH EXPRESSION;Cf;0;S;;;;;N;;;;;
0081;END OF MATH EXPRESSION;Cf;0;S;;;;;N;;;;;
0082;LINE BREAK PERMITTED HERE;Cf;0;BN;200B;;;;N;BREAK PERMITTED HERE;;;;
0083;LINE BREAK NOT PERMITTED HERE;Cf;0;BN;2060;;;;N;NO BREAK HERE;;;;
0084;MATH HORIZONTAL DIVISION;Sm;0;S;;;;;N;INDEX;;;;
0085;NEXT LINE;Zp;0;B;000D 000A;;;;N;;;;;   Decomposes to CRLF; Note: must be composition excluded!
0086;INFORMATION LEFT BRACKET ONE;Cf;0;B;;;;;N;START OF SELECTED AREA (block terminal
      semantics);;;;         generalised
0087;INFORMATION RIGHT BRACKET ONE;Cf;0;B;;;;;N;END OF SELECTED AREA (block terminal
      semantics);;;;         generalised
0088;<reserved>;Cn;0;BN;;;;;N;CHARACTER TABULATION SET (HORIZONTAL TABULATION SET);;;;
0089;CHARACTER TABULATION WITH JUSTIFICATION;Cf;0;S;;;;;N;HORIZONTAL TABULATION WITH
      JUSTIFICATION;;;;      HT+move the text of the preceding 'field' (between HT(J)s/l.b.) to end at the end of the 'field'
008A;<reserved>;Cn;0;BN;;;;;N;LINE TABULATION SET (VERTICAL TABULATION SET);;;;
008B;PARTIAL LINE FORWARD;Cf;0;S;;;;;N;PARTIAL LINE DOWN;;;;
008C;PARTIAL LINE BACKWARD;Cf;0;S;;;;;N;PARTIAL LINE UP;;;;
008D;REVERSE LINE FEED;Zl;0;S;;;;;N;REVERSE INDEX;;;;
008E;<reserved>;Cn;0;BN;;;;N;SINGLE SHIFT TWO;;;;
008F;<reserved>;Cn;0;BN;;;;;N;SINGLE SHIFT THREE;;;;
0090;DEVICE CONTROL STRING;Cf;0;BN;;;;;N;;;;;    has been used for some terminals (outdated) (Cn?)

0091;INFORMATION SEPARATOR FIVE;Cf;0;B;;;;;N;PRIVATE USE ONE;;;;     still private use
0092;INFORMATION SEPARATOR SIX;Cf;0;B;;;;;N;PRIVATE USE TWO;;;;      still private use
```

```
0093;<undefined>;Cn;0;BN;;;;;N;SET TRANSMIT STATE (defunct);;;;        left unassigned for now
0094;BACKSPACE;Cf;0;BN;;;;;N;CANCEL CHARACTER (moved);;;;              swapped to here; not defunct
0095;<undefined>;Cn;0;BN;;;;;N;MESSAGE WAITING (defunct);;;;           left unassigned for now
0096;INFORMATION LEFT BRACKET TWO;Cf;0;B;;;;;N;START OF GUARDED AREA (START OF PROTECTED
      AREA) (block terminal semantics);;;;                 generalised
0097;INFORMATION RIGHT BRACKET TWO;Cf;0;B;;;;;N;END OF GUARDED AREA (END OF PROTECTED
      AREA) (block terminal semantics);;;;                 generalised
0098;START OF STRING;Cf;0;BN;;;;;N;;;;;                    see main text
0099;MATH MIRROR SYMBOL;Cf;0;BN;;;;;N;;;;;
009A;SINGLE CHARACTER INTRODUCER;Cf;0;BN;;;;;N;;;;;        this with a follow-on char creates another control
009B;CONTROL SEQUENCE INTRODUCER;Cf;0;BN;;;;;N;;;;;        see main text for how this creates new controls
009C;STRING TERMINATOR;Cf;0;BN;;;;;N;;;;;                  see main text
009D;OPERATING SYSTEM COMMAND;Cf;0;BN;;;;;N;;;;;           see main text
009E;PRIVACY MESSAGE;Cf;0;BN;;;;;N;;;;;    message text for a status line in some terminals (outdated) (Cn?)
009F;APPLICATION PROGRAM COMMAND;Cf;0;BN;;;;;N;;;;;        see main text

2028;LINE SEPARATOR;Zl;0;S;;;;;N;;;;;
```

“Non-characters”, primary use: object replacement (must **not** be deleted in bidi processing):

```
FDD0;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDD1;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDD2;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDD3;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDD4;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDD5;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDD6;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDD7;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDD8;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDD9;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDDA;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDDB;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDDC;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDDD;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDDE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDDF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDE0;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDE1;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDE2;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDE3;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDE4;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDE5;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDE6;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDE7;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDE8;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDE9;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDEA;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDEB;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDEC;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDED;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDEE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FDEF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;

FFFE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FFFF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
1FFFE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
1FFFF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
2FFFE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
2FFFF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
3FFFE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
3FFFF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
4FFFE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
4FFFF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
5FFFE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
5FFFF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
6FFFE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
6FFFF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
7FFFE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
7FFFF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
8FFFE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
8FFFF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
9FFFE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
9FFFF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
AFFFE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
AFFFF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
BFFFE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
BFFFF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
CFFFE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
CFFFF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
DFFFE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
DFFFF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
EFFFE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
EFFFF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FFFFE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
FFFFF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
10FFFE;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
10FFFF;<OBJECT REPLACEMENT>;So;0;ON;;;;;N;;;;;
```

Appendix D: C0/C1/nonch for LineBreak.txt

C0 and U+007F:

```
0000           ; CB # Cn        <reserved-0000>        permanently
0001..0003     ; BK # Zp     [3] START OF HEADING..END OF TEXT
0004..0008     ; CM # Cd     [5] END OF TRANSMISSION..CANCEL CHARACTER
0009           ; BA # Cf         CHARACTER TABULATION
000A           ; LF # Zp         LINE FEED
000B..000C     ; LF # Zl     [2] LINE TABULATION..FORM FEED
000D           ; CR # Zp         CARRIAGE RETURN    Zp iff not immediately followed by LF/VT/FF nor a filler
000E..000F     ; CM # Cn     [2] <reserved-000E>..<reserved-000F>     permanently
0010..0015     ; CM # Cd     [6] DEVICE CONTROL ZERO..DEVICE CONTROL FIVE
0016           ; CB # Cf         INFORMATION SEPARATOR ZERO
0017..0019     ; CM # Cd     [3] DEVICE CONTROL EIGHT..DEVICE CONTROL NINE
001A           ; AI # So         SUBSTITUTE
001B..001F     ; CB # Cf     [5] ESCAPE..INFORMATION SEPARATOR ONE
007F           ; CM # Cd         DEVICE CONTROL TEN
```

C1:

```
0080..0081     ; CM # Cf     [2] START OF MATH EXPRESSION..END OF MATH EXPRESSION
0082           ; ZW # Cf         LINE BREAK PERMITTED HERE
0083           ; WJ # Cf         LINE BREAK NOT PERMITTED HERE
0084           ; BK # Sm         MATH HORIZONTAL DIVISION
0085           ; BK # Zp         NEXT LINE             Note that NL ≡ BK, so we use BK here, NL is redundant
0086..0087     ; CB # Cf     [2] INFORMATION LEFT BRACKET ONE..INFORMATION RIGHT BRACKET ONE
0088           ; XX # Cn         <reserved-0088>
0089           ; BA # Cf         CHARACTER TABULATION WITH JUSTIFICATION
008A           ; XX # Cn         <reserved-008A>
008B..008C     ; CM # Cf     [2] PARTIAL LINE FORWARD..PARTIAL LINE BACKWARD
008D           ; BK # Zl         REVERSE LINE FEED
008E..008F     ; CM # Cn     [2] <reserved-008E>..<reserved-008F>     permanently
0090           ; CM # Cf         DEVICE CONTROL STRING
0091..0092     ; CB # Cf     [2] INFORMATION SEPARATOR FIVE..INFORMATION SEPARATOR SIX
0093           ; XX # Cn         <undefined-0093>
0094           ; CM # Cf         BACKSPACE
0095           ; XX # Cn         <undefined-0095>
0096..0097     ; CB # Cf     [2] INFORMATION LEFT BRACKET TWO..INFORMATION RIGHT BRACKET TWO
0098..009E     ; CM # Cf     [7] START OF STRING..PRIVACY MESSAGE
009F           ; CB # Cf         APPLICATION PROGRAM COMMAND
```

“Non-characters”, primary use: object replacement (must **not** be deleted in bidi processing):

```
# “non-characters”, but actual use would primarily be as object replacement characters:
FDD0..FDEF     ; CB # So    [32] <OBJECT REPLACEMENT-FDD0>..<OBJECT REPLACEMENT-FDEF>

# “non-characters”, but actual use would primarily be as object replacement characters:
FFFE..FFFF     ; CB # So     [2] <OBJECT REPLACEMENT-FFFE>..<OBJECT REPLACEMENT-FFFF>
1FFFE..1FFFF   ; CB # So     [2] <OBJECT REPLACEMENT-1FFFE>..<OBJECT REPLACEMENT-1FFFF>
2FFFE..2FFFF   ; CB # So     [2] <OBJECT REPLACEMENT-2FFFE>..<OBJECT REPLACEMENT-2FFFF>
3FFFE..3FFFF   ; CB # So     [2] <OBJECT REPLACEMENT-3FFFE>..<OBJECT REPLACEMENT-3FFFF>
4FFFE..4FFFF   ; CB # So     [2] <OBJECT REPLACEMENT-4FFFE>..<OBJECT REPLACEMENT-4FFFF>
5FFFE..5FFFF   ; CB # So     [2] <OBJECT REPLACEMENT-5FFFE>..<OBJECT REPLACEMENT-5FFFF>
6FFFE..6FFFF   ; CB # So     [2] <OBJECT REPLACEMENT-6FFFE>..<OBJECT REPLACEMENT-6FFFF>
7FFFE..7FFFF   ; CB # So     [2] <OBJECT REPLACEMENT-7FFFE>..<OBJECT REPLACEMENT-7FFFF>
8FFFE..8FFFF   ; CB # So     [2] <OBJECT REPLACEMENT-8FFFE>..<OBJECT REPLACEMENT-8FFFF>
9FFFE..9FFFF   ; CB # So     [2] <OBJECT REPLACEMENT-9FFFE>..<OBJECT REPLACEMENT-9FFFF>
AFFFE..AFFFF   ; CB # So     [2] <OBJECT REPLACEMENT-AFFFE>..<OBJECT REPLACEMENT-AFFFF>
BFFFE..BFFFF   ; CB # So     [2] <OBJECT REPLACEMENT-BFFFE>..<OBJECT REPLACEMENT-BFFFF>
CFFFE..CFFFF   ; CB # So     [2] <OBJECT REPLACEMENT-CFFFE>..<OBJECT REPLACEMENT-CFFFF>
DFFFE..DFFFF   ; CB # So     [2] <OBJECT REPLACEMENT-DFFFE>..<OBJECT REPLACEMENT-DFFFF>
EFFFE..EFFFF   ; CB # So     [2] <OBJECT REPLACEMENT-EFFFE>..<OBJECT REPLACEMENT-EFFFF>
FFFFE..FFFFF   ; CB # So     [2] <OBJECT REPLACEMENT-FFFFE>..<OBJECT REPLACEMENT-FFFFF>
10FFFE..10FFFF ; CB # So     [2] <OBJECT REPLACEMENT-10FFFE>..<OBJECT REPLACEMENT-10FFFF>
```

Appendix E: Device control characters for tty “cooked”/“sane” Unix/Linux tty mode

This appendix looks a bit closer at the device control characters (Cd == Cc) in the C0 space.

1) We must avoid C0/C1 characters that are used for data. For instance, ETX can be used in math expressions (and in a form of “telegrams”, which was the original intent, but now defunct), and avoid **IS***n* that can also be part of data; especially when pasted in (these are not so likely typed in manually directly).
2) We also must avoid code switching characters like LS0.
3) While DEVICE CONTROL STRING and PRIVACY MESSAGE start what can be regarded as device ‘setting’ control strings, these characters are not device controls by themselves.
4) Try to stay close to original semantics for controls that we use as device controls. Obviously DEVICE CONTROL n (which are formally “private use”) can be used.
5) Avoid using ctrl-<something> that is commonly used for something else, in particular ctrl-c and ctrl-v. These two we would like to use for “local copy-and-paste”; ctrl-c and ctrl-v respectively; ‘local’ referring to that ctrl-c and ctrl-v are executed locally , i.e. in the terminal emulator, not by the remote system. One should still have easy access to ‘remote edits’ and ‘remote signal generation (for a few SIGnals).
6) Try to use more intuitive ctrl-<something> when possible.
7) There is basically only one device control that has meaning when sent **to** a terminal: ALERT BELL. Sent from a terminal it should be regarded as an error.
8) All other device controls have (or might have) meaning only when sent from a terminal emulator to a remote system.
9) We will to consider DATA LINK ESCAPE as highly defunct.
10) Device controls sent *from* the terminal emulator to the remote system system, that when sent *to* the terminal all, except (ALERT BELL), has no meaning and are indeed undefined (display as SUBSTITUTE/REPLACEMENT CHARACTER or other error indication).
11) Setting of ctrl-c and ctrl-v (and possibly others) must be done in the terminal emulator; some terminal emulators already allow this; while still not losing any of the (very limited) editing command done remotely.
12) The changes are intended to make for a better user experience, users being used to that ctrl-c means copy and ctrl-q means *orderly terminate the target program*.
13) We introduce one “new” general category, Cd, for device control characters. Since the general categories in Unicode are fixed, we actually have this an alias for Cc, but with emphasises on the device control nature. All other Cc (the entire C0 and C1 space) have been assigned other general categories (and revised bidi and line break categories).
14) The following table gives the device control characters, and two Zp characters that are not device controls, but still (can) cause flushing of the tty line buffer after being included at the end of the tty line buffer. (Several “command line” programs, like ‘bash’, use raw tty mode, and do their own command line editing, including interpreting cursor movement control sequences, which the tty’s (until now at least) do not interpret, as well as their own interpretation (or, usually, not) of process controlling Cd chars.)
15) Note that all characters that are not marked as Cd here are *data characters*, and may reasonably occur in text data that is pasted in to a tty via a terminal emulator (and elsewhere).

c.p.	^	Name/comment (note that these apply (approximately) to tty “cooked” mode, not otherwise)	g.c.	Bidi	LB
0004	**d**	END OF TRANSMISSION *(use for (stty)* ***eof;*** *used to* ***flush the tty (cooked mode) input line buffer*** *without the eof character. In contrast to the eol chars, the eof char is not propagated to the reading program. If the line buffer is empty, returns an empty string from ‘read’, and that is often interpreted as EOF; however, there is nothing preventing another read (from the same tty or actual file) and that read may return some non-empty string; so no file close, i.e. eof is NOT actually ending the input from the tty). In “raw” mode, the eof character is just forwarded to the reading program.*	**Cd**	**B**	**CM**
0005	**e**	~~ENQUIRY~~ **“DEVICE CONTROL SIX”**	**Cd**	**BN**	**CM**
0006	**f**	~~ACKNOWLEDGE~~ **“DEVICE CONTROL SEVEN”** *(can be used for* ***lnext; not*** *POSIX)*	**Cd**	**BN**	**CM**
0007	**g**	**ALERT** BELL *(brief audible or visible alert to user)* *(has no meaning when sent to the “remote” side, however, it may be echoed* *and then it will be received back to the terminal)* *Note that this character is renamed to ALERT BELL in order to have a different name from U+1F514 BELL; cmp. \a for ‘alert’.*	**Cd**	**BN**	**CM**
0008	**h**	CANCEL CHARACTER *(was BACKSPACE, not erasing);* *de facto tty use is CANCEL CHARACTER (erasing)* *(note that this erases only within the current tty line buffer;* *in addition, erasing an HT might not be properly reflected in the terminal emulator display; also, if the terminal emulator side has done automatic line breaks, there may also be display glitches)*	**Cd**	**BN**	**CM**
000A	**j**	LINE FEED *(****eol0****, flush input line buffer in tty after getting this char.; Unix/Linux tty cooked mode* ***output:*** *mapped to CRLF)*	**Zp**	**B**	**LF**
000D	**m**	CARRIAGE RETURN *(Unix/Linux tty cooked mode* ***input:*** *mapped to LF) (CR unmapped, stty -icrnl; use for* ***eol2****, second eol char)*	**Zp**	**B**	**CR**
0010	**p**	~~*DATA LINK ESCAPE*~~ **“DEVICE CONTROL ZERO”**	**Cd**	**BN**	**CM**
0011	**q**	DEVICE CONTROL ONE *(is commonly used for* ***XON****/****start****)*	**Cd**	**BN**	**CM**
0012	**r**	DEVICE CONTROL TWO *(is commonly used for* ***rprnt;*** *reprint of current tty input line buffer;* ***not*** *POSIX)*	**Cd**	**BN**	**CM**
0013	**s**	DEVICE CONTROL THREE *(is commonly used for* ***XOFF/stop****)*	**Cd**	**BN**	**CM**
0014	**t**	DEVICE CONTROL FOUR *(can be used for* ***SIGINT****/****intr****; terminate)*	**Cd**	**BN**	**CM**
0015	**u**	~~NEGATIVE ACKNOWLEDGE~~ **“DEVICE CONTROL FIVE”** *(can be used for* ***SIGQUIT/quit*** *terminate with core dump)*	**Cd**	**BN**	**CM**
0017	**w**	~~END OF TRANSMISSION BLOCK~~ **“DEVICE CONTROL EIGHT”** *(is used for* ***werase****, “CANCEL WORD”;* ***not*** *POSIX)*	**Cd**	**BN**	**CM**
0018	**x**	CANCEL [LINE] *(should be used for* ***kill;*** *erase the current tty input line)*	**Cd**	**BN**	**CM**
0019	**y**	~~END OF MEDIUM~~ **“DEVICE CONTROL NINE”** *(can be used for* ***SIGTSTP****/****susp****)*	**Cd**	**BN**	**CM**
007F	**?**	~~DELETE~~ *(was eliminated)* **“DEVICE CONTROL TEN”**	**Cd**	**BN**	**CM**

Except for eol/eol2 (which just trigger tty line buffer flush as an *additional* effect) no non-Cd character (like Zp, Zl or Cn code points, as updated here in the appendices) should be used for tty device controls. In particular Zp characters which may occur as parts of pasted in strings, like original CSV format or math expressions. While maybe not easily typed, they may well be pasted in as input also to terminal based programs. Then having them trigger some device control would not be great.

This can be summarised to the following *stty* command (for interactive *bash*):

```
case "$-" in  # check if the standard input is a tty before setting cooked ctrls
*i*)  # it is a tty; note that ^j (LF) is fixed, and NEL cannot be set if UTF-8
  stty swtch ^e  lnext ^f  eof ^d eol ^m eol2 ^-  rprnt ^r  start ^q stop ^s \
    discard ^? susp ^y dsusp ^p intr ^t quit ^u  erase ^h werase ^w kill ^x  ;;
esac  # note that these dev ctrls can work only via a tty, file reading is ‘raw’
```

Note that the various Unix/Linux/MacOS are not exactly equal on this point; check and test on the relevant system. In particular, some of the “control functionality” might not be implemented, especially the non-POSIX ones are often not implemented.

The command here includes some options that are not available on all system variants. This is a suggested recommended assignment; it is *not* part of this proposal for ECMA-48 update per se.

Unfortunately, the common default for these tty settings use “control” characters that are not of general category Cd (as defined in appendices B to D), and are unsuitable as device controls for that reason and for other reasons.

Note that the tty line editing is done only in so-called ‘cooked’ mode for the tty, which is suitable for many ‘naïve’ terminal-based programs. However, many systems programs, like ‘bash’ and other command interpreters, do their own command line editing and operate in tty “raw” mode which implies no tty line editing. That enables such things as command line history and tab completion (in ‘bash’: of file names, but in other programs it can be something else, like command completion). For ease of use, these programs should interpret at least CANCEL CHARACTER as a character eraser, and CANCEL LINE as a (command) line eraser (in ‘raw’ tty mode these characters are passed on to the target program as is). Note that in ‘raw’ mode CR is not mapped (by the tty) to LF (Unix/…).

This way we have 1) avoided all data characters (like IS4, ETX, …) as device controls (these may now be pasted (with ctrl-v) in without interference with device controls, like math expressions using ETX or CVS- or JSON-like data using ISn), 2) avoided all “never use” “ex-characters” as device controls, 3) allow ctrl-c, ctrl-v to be used locally by the terminal emulator in their copy-and-paste use without blocking device controls over the tty.

These actions are for “cooked” mode (for ‘line’ batch input) via tty. *emacs*, *bash*, and some other programs use “raw” mode tty reading (and read character (byte, actually) by character), and do their own “line editing” (or similar). Those programs are not affected by “cooked” mode tty settings. Note that “raw” mode is not easily handled, there is not even system support for “EOF” (instead EOT is sent to the application), \r and \n have “device control” meanings on output to a terminal emulator, \r is not automatically mapped to \n on input, and there is no way to generate SIGINT from the keyboard via the operating system. So most programs use “cooked” mode for tty input/output, but with significant exceptions such as emacs, vim, less, bash and other programs.

Appendix F: INFORMATION SEPARATOR revival

CSV (in multiple variants) along with JSON are popular data storage formats, especially for moving data from one application to another. But CSV has a predecessor, which actually was a better format than the current CSV. It is the UNIT SEPARATOR/RECORD SEPARATOR/GROUP SEPARATOR/FILE SEPARATOR format of EBCDIC and ASCII; replaced by the more generic INFORMATION SEPARATORs in ECMA-48 (the US/RS/GS/FS were never in ECMA-48). Now, this predecessor of CSV actually has some advantages over current CSV: apart from that the predecessor allows for multiple tables in sequence (which we will gloss over, since that is not the good part), it allows for not needing special syntax for character that are used for syntax in current CSV (comma, tab, semicolon, space, newline) when they occur as data. That is because the INFORMATION SEPARATORs must never occur as data (if they do, there is a coding error, replace by SUB), and thus no need for any quoting mechanism (\, "") for syntax characters in data. And for current CSV, syntax characters (comma, semicolon, ", tab, space, newline) can be quite common in the actual data.

IS1 to **IS4** are the more flexible *replacements* (*no* equivalence) for UNIT SEPARATOR to FILE SEPARATOR from ASCII/EBCDIC; this is the *original* CSV data format. They can be used for a CSV-like format or a JSON-like format, where instead of comma, newline, etc., one can use **IS1**, …, **IS6**, for the data table syntax. The ISn themselves shall be visibly rendered in display.

Use (e.g.) **IS1** instead of ‘,’ and **IS2** instead of \n for a CSV format (.isv…). Then one does *not* need to use character references (like “\,”, “\n”, …) or to use quotes ("…") where comma (or semicolon, tab, …), newline, etc. appear in the actual data.

IS1 to **IS4**, in contrast to **US** to **FS** in ASCII, do *not* have a *predetermined* hierarchy (like operator precedence), indeed there need be no hierarchy between them. A JSON-*like* format can use: **ILB1**(**ESC F**) for ‘{‘, **IRB1**(**ESC G**) for ‘}’, **IS5**(**ESC Q**) for ‘:’, **IS6**(**ESC R**) for ‘,’, **ILB2**(**ESC V**) for ‘[‘, **IRB2**(**ESC W**) for ‘]’. This ‘mapping’ avoids **IS1** to **IS4**, and thus *allows* for **IS1**..**IS3** based CSV data as ‘value’ in this example JSON-ish format. Note that there is no need for quotes or character references, since the **IS***n*:s must not be part of actual data (if they are, replace by **SUB** since it is an error). Note that here we use the **INFORMATION LEFT**/**RIGHT BRACKET ONE**/**TWO** control codes. *Another* example of (private!) use of these four characters is the original **START**/**END OF SELECTED**/**GUARDED AREA** (here generalised to be ‘private use’) for “block” terminals, in case one wants to emulate this kind of (very old-fashioned) terminals.

These formats are only hints; ***not** part of this proposed update*, just mentioning possibilities where these characters still can be put to good use and that is less error prone than ordinary CVS and JSON; no need to add quotes or the like, as well as make it easier to prevent data structure injection (just check that there are no **IS***n* in the stored data; if there is, replace by **SUB**). When displaying the source of **IS***n* based source data, there is less trouble with bidi, since data cells are bidi isolated (since the**IS***n* have bidi category B, as given in this proposal). So basing a CVS-like/JSON-like/similar data representation structure on **IS***n* has *multiple advantages* over using “,”, “{“, etc., characters not reserved for data structure syntax, for the structure syntax.

We can also handle bibliographic sorting information separators (detailed syntax, such as nestability and order, is not known), transferring what was registered in https://www.itscj-ipsj.jp/ir/124.pdf (no longer accessible), using the INFORMATION SEPARATORs, and use SCI to mark “end of”. See the following table, where also other possible representations are given, avoiding the **defunct** “C0/C1 control codes registry”.

Note: The “C0/C1 control codes registry” is defunct since: a) it was a bad idea to begin with, b) nobody is using it, in particular nobody is using the code switching escape sequences, c) the registry is filled with garbage (except possibly the latest one for bibliographic use), and d) the registry has not been maintained in decades, and now also the web site is gone, and, most importantly, e) implementing any switching as suggested for this registry would be a complete disaster for the entire modern IT infrastructure.

reg. 124	Mnem	Name	‘printable’	p.u.	Info.sep.(p.u.)
87	CUS	Close-up for sorting	COMBINING GRAPHEME JOINER (U+034F)		
88	NSB	Non-sorting characters begin	❰ (U+2770)	U+F80A	**IS1**
89	NSE	Non-sorting characters end	❱ (U+2771)	U+F80B	**SCI IS1**
8B	PLD	Partial line down	**PLD** (U+008B) (But better to use ECMA-48 superscript/subscript styling per prop. extension; **CSI 56:1m** … **CSI 56:0m**; **CSI 56:=1m** … **CSI 56:0m**)		
8C	PLU	Partial line up	**PLU** (U+008C) (But better to use ECMA-48 superscript/subscript styling per prop. extension; **CSI 56:1m** … **CSI 56:0m**; **CSI 56:=1m** … **CSI 56:0m**)		
91	EAB	Embedded annotation begin	⟬ (U+27EC)	U+F800	**IS2**
92	EAE	Embedded annotation end	⟭ (U+27ED)	U+F801	**SCI IS2**
95	SIB	Sorting interpolation begin	⦃ (U+2983)	U+F802	**IS3**
96	SIE	Sorting interpolation end	⦄ (U+2984)	U+F803	**SCI IS3**
97	SSB	Secondary sorting value begin	⦅ (U+2985)	U+F804	**IS4**
98	SSE	Secondary sorting value end	⦆ (U+2986)	U+F805	**SCI IS4**
9C	KWB	Keyword begin	⦉ (U+2989)	U+F806	**IS5**
9D	KWE	Keyword end	⦊ (U+298A)	U+F807	**SCI IS5**
9E	PSB	Permutation string begin	⦑ (U+2991)	U+F808	**IS6**
9F	PSE	Permutation string end	⦒ (U+2992)	U+F809	**SCI IS6**

Indeed, we can even have an IS based format that is like XML: use **INFORMATION LEFT**/**RIGHT BRACKET ONE** for ‘<’ and ‘>’ of a begin tag, **INFORMATION LEFT**/**RIGHT BRACKET TWO** for ‘</’ and ‘>’ of an end tag, **IS5** for ‘=’, **IS6** for a space between attributes. There is no need for quote marks, nor for ‘&lt;’, ‘&quot;’ or other delayed character references (and thus not need for using an IS corresponding to &). One can even have **IS1**..**IS3** based CSV data between XML-ish start/end tags.

Note that all of these are examples for illustration of INFORMATION SEPARATORs. They are *not* part of this proposal for ECMA-48 update. There is also an example of the (private use) use of **SCI** *outside* of math expressions (otherwise undefined…). *Inside* of math expressions, **SCI** has specified uses, see separate proposal (not part of this ECMA-48 update).

Appendix H: Notes on some bidi issues

The bidi algorithm is far from perfect. Here we discuss some of the issues that may arise.

Bidi issues with "phrase" punctuation

Quote marks as well as open/close parentheses/brackets/… all are bidi neutral. So are end phrase punctuation (FULL STOP, COMMA, SEMICOLON, …). This means that if there is a switch in strong bidi directionality of the characters close these punctuation marks, the bidi algorithm may assign the wrong directionality to these (originally) bidi neutral marks. And that does happen, resulting in incorrect placement of the marks and thus incorrect reading of the text, as displayed for a human reader or an "OCR reader".

Therefore, in case of bidi directional switch in the immediate vicinity of any of the phrase start or end punctuation, add LEFT-TO-RIGHT MARK or RIGHT-TO-LEFT MARK (as appropriate) adjacent to the punctuation mark(s) so that the punctuation mark(s) get the directionality of the *surrounding* text (not the inner text), to avoid that the punctuation mark is misplaced by the bidi algorithm.

A text editor may help to semi-automatically insert those when needed, so that the (human…) author gets some help in inserting them. Interestingly, adding line breaks (LS or VT) can also help, but we do not want to sprinkle in line breaks when not appropriate. (The nesting bidi controls, LRI…PDI, RLI…PDI, LRO…PDF, RLO…PDF, are rarely or even very rarely advisable. However, LRI…PDI or RLI…PDI, including aligning the line start position appropriately, can sometimes be useful for *full paragraphs*.)

Bidi issues with other bidi neutral characters

Also other symbols/punctuation (that is bidi neutral to begin with) may get the wrong directionality assigned to them by the bidi algorithm in case there is a switch of strong bidi directionality in their immediate vicinity. In this case, use the LEFT-TO-RIGHT MARK or RIGHT-TO-LEFT MARK adjacent to the punctuation/symbol, so that it gets the desired directionality, i.e., so that there is no switch of strong directionality in the immediate vicinity of the symbol/punctuation. Again, a text editor may help suggesting to insert those bidi marks when needed, though there is no direct clue (at the character level) to which directionality it should belong. Another complication is that some "phrase" punctuation, like ")", ".", may sometimes be used in a "non-phrase" way, like here, when discussing about the character, or using "." (normally phrase ending) inside a file name (and then it is not phrase ending).

Bidi issues in programming and data languages as well as math expressions

Just to emphasise (as we have exposed this already), for programming languages and data languages (CSV, JSON, XML, …) (i.e. the "source code", as edited in e.g. IDE editors) as well as math expressions, it is essential *not* to use the "general" bidi algorithm. If any bidi algorithm is at all used in these cases, it *must* be limited to "identifiers" and numeric literals (numerals), e*ach one in isolation* (though no bidi control characters).