L2/23-107

Proper Complex Script Support in Text Terminals.

Renzhi LI, Dustin HOWETT, Peter CONSTABLE

Summary

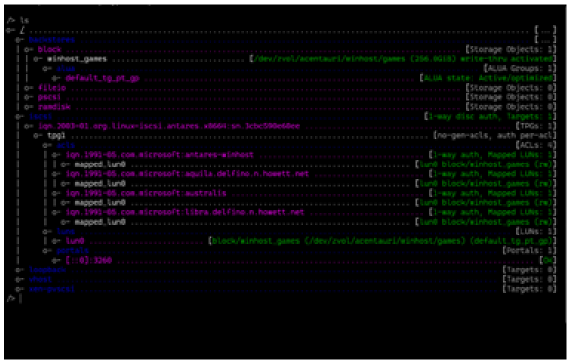
Terminal environments and text-based command-line interface (CLI) applications are typically associated with simple text layout implementations that are not capable of supporting many scripts, such as Arabic or Devanagari.

This document proposes a new Unicode project to develop specifications (*Terminal Complex Script Support*, or *TCSS*) that will allow terminal environments and CLIs to provide comprehensive support for Unicode scripts. These specifications could eventually be published as a new UTS or other Unicode publication. Standardization of such specifications would facilitate interoperability between different CLIs and terminal environments. We will outline TCSS' main requirements and provide examples for them.

Our goal is to start a project in the Unicode Technical Committee (UTC) to create a detailed specification for TCSS, and if necessary, coordinate with other standardization organizations to standardize any parts of TCSS that might lie outside UTC's scope.

Introduction

Command-line interface (CLI) and Text-based user interface (TUI) applications play important roles in computer systems, especially for software developers and system administrators. When operating such applications, users use *text terminals* to interact with them.



```
/> ls
o- / ......................................................................... [...]
  o- backstores .............................................................. [...]
  | o- block ................................................. [Storage Objects: 1]
  | | o- winhost_games ........... [/dev/zvol/acentauri/winhost/games (256.0GiB) write-thru activated]
  | |   o- alua .................................................... [ALUA Groups: 1]
  | |     o- default_tg_pt_gp ........................ [ALUA state: Active/optimized]
  | o- fileio ................................................ [Storage Objects: 0]
  | o- pscsi ................................................. [Storage Objects: 0]
  | o- ramdisk ............................................... [Storage Objects: 0]
  o- iscsi ............................................ [1-way disc auth, Targets: 1]
  | o- iqn.2003-01.org.linux-iscsi.antares.x8664:sn.3cbc590e60ee .............. [TPGs: 1]
  |   o- tpg1 ............................................ [no-gen-acls, auth per-acl]
  |     o- acls ................................................................ [ACLs: 4]
  |     | o- iqn.1991-05.com.microsoft:antares-winhost ........ [1-way auth, Mapped LUNs: 1]
  |     | | o- mapped_lun0 ........................... [lun0 block/winhost_games (rw)]
  |     | o- iqn.1991-05.com.microsoft:aquila.delfino.n.howett.net [1-way auth, Mapped LUNs: 1]
  |     | | o- mapped_lun0 ........................... [lun0 block/winhost_games (rw)]
  |     | o- iqn.1991-05.com.microsoft:australis .............. [1-way auth, Mapped LUNs: 1]
  |     | | o- mapped_lun0 ........................... [lun0 block/winhost_games (rw)]
  |     | o- iqn.1991-05.com.microsoft:libra.delfino.n.howett.net [1-way auth, Mapped LUNs: 1]
  |     |   o- mapped_lun0 ........................... [lun0 block/winhost_games (rw)]
  |     o- luns ................................................................ [LUNs: 1]
  |     | o- lun0 ......... [block/winhost_games (/dev/zvol/acentauri/winhost/games) (default_tg_pt_gp)]
  |     o- portals .......................................................... [Portals: 1]
  |       o- [::0]:3260 ............................................................ [OK]
  o- loopback ........................................................... [Targets: 0]
  o- vhost .............................................................. [Targets: 0]
  o- xen-pvscsi ......................................................... [Targets: 0]
/> 
```

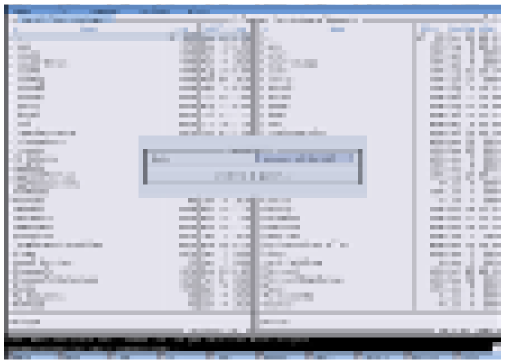


Figure 1 Example of modern CLI and TUI application

A "terminal", in general, is a hardware device or piece of software that can be used for entering data into and transcribing data from a computer or a computing system. There are many types of computer terminals that have existed in history and currently exist in modern industry, but the most common subtype is the text terminal, which operate over text streams. It is this type of terminal that we will primarily refer to throughout this document.

For a long time, text terminals only supported European writing systems, mainly Latin. Starting in the 1980s, they were adopted in East Asia and gained support for CJK (Chinese, Japanese and Korean) writing systems. However, even as of today (2023), most text terminals still lack proper support for more complicated writing systems, like Devanagari and Arabic, which makes them difficult to interact with for almost half of the world's population. Compared to other text-related applications such as word processors, typesetting software and web browsers, text terminals have fallen far behind the rest of the industry.

Features of a Text Terminal

Text terminals descended from teletypewriter ("TTY", or "teletype") machines, which produced immutable hard copy and had a single unidirectional "print head". When connected to a computer, a teletype will accept user input from its keyboard and print out the text it received from the computer on a roll of paper. Teletypes provided one of the earliest user interfaces for computers.

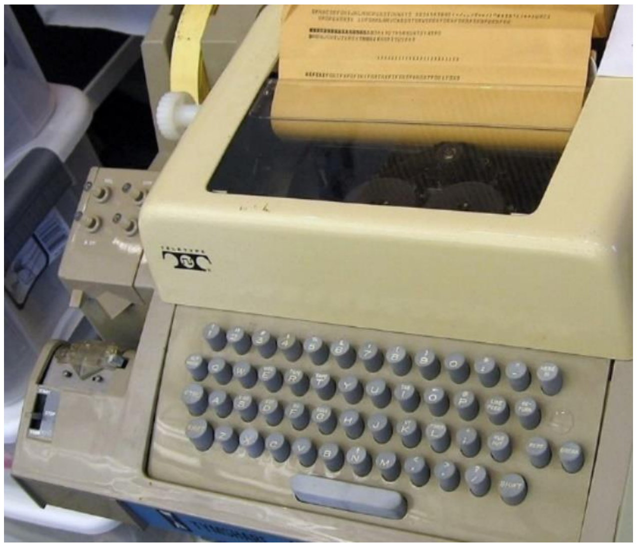


Figure 2 Teletype Model 33, a widely used teletype used as a computer terminal.

Later, text terminals equipped with a video display unit (such as a CRT screen) became widely adopted by computer operators. Early models of video text terminals usually emulated the teletypes, allowing them to replace teletype machines as part of cost-saving measures.

Starting from the 1970s, video text terminals like Lear Siegler's ADM-3A, DEC VT52 and DEC VT100 started to go beyond teletype emulation with many more features. The "print head" of yore became addressable, and this gave them the ability to freely print information at any location on the screen. These devices were closely intertwined with the era of time-sharing minicomputers. Many hardware terminals support specialized control sequences to manipulate the screen display and input, some of which became standardized as "ANSI" (American National Standards Institute) escape sequences.[1]

The basic operation remains the same, however: terminals receive a stream of text data for presentation to a user, who can act on it and respond via an input device, which sends a text stream back to the computer connected to the terminal.

Figure 3 ADM-3A, a famous video text terminal used in 1970s.

Starting in the late 1980s, as a side effect of increasing use of graphical user interfaces, software-implemented text terminal emulators gradually replaced hardware terminals. Famous examples of terminal emulators include *xterm* on Unix-like systems, iTerm2 on Macintosh, and Windows Terminal on Windows.

Because of that history, there are a few implementation details that are conserved in both hardware terminals and software terminal emulators:

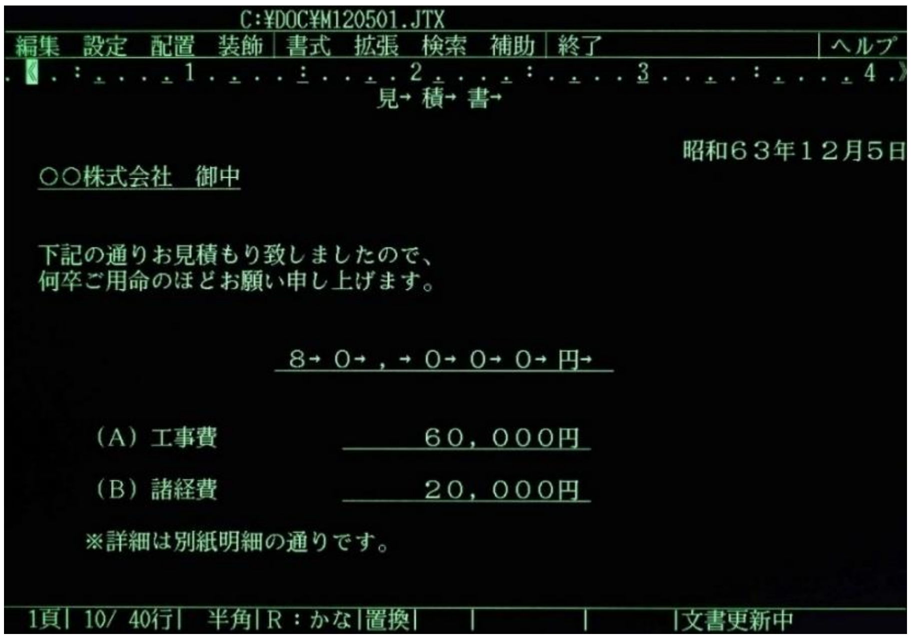
- There is a text surface, *the screen*, that is presented to the user.
- Text displayed on the screen will be aligned to a *grid*, which is divided into *rows* and *columns*. The intersection of one column and one row is called a *cell.*
- The terminal will receive another process' or another machine's output as a text stream. The output stream contains either control sequences to be consumed by the terminal to perform specific operations, or graphic characters to be displayed on the screen.
- There is a *cursor* representing the point at which incoming text will be written. When text is written, it overstrikes the existing content in the cell(s) at the cursor position.

[1] Escape control sequences were standardized by ANSI in 1979 as ANSI X3.64. Other related standards include ECMA-48 and ISO/IEC 6429. In addition to standardized functions of individual character codes, such as CARRIAGE RETURN, terminals and CLIs use control sequences to achieve certain behaviors, such as **ESC [*n* D**, which moves the cursor back by *n* cells.

- Applications on the “other end” of a terminal session cannot easily determine the terminal’s preferred text layout parameters, as there are only limited mechanisms for querying those parameters.

In the pre-Unicode era, text layout in terminals was straightforward to implement. As applications were just handling Latin, they could simply map one character to one screen cell. However, this simple solution becomes problematic in the face of a larger supported character set.

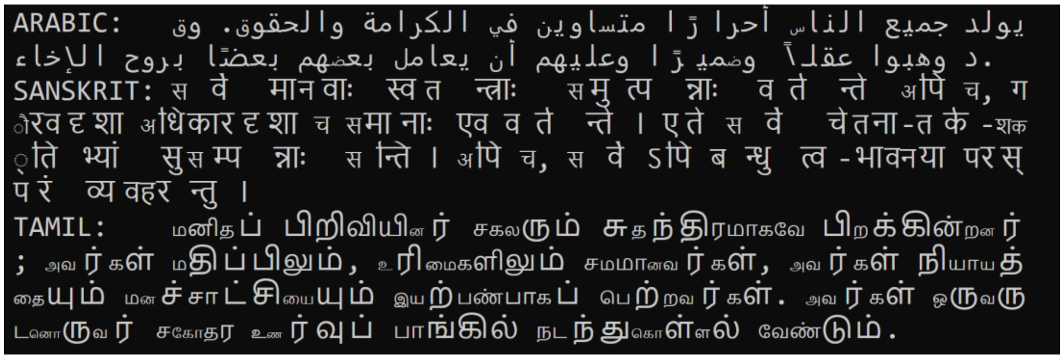
East-Asian countries adopted computers shortly after the western hemisphere. The problem they face is the large character sets they require. Their early character sets like Shift-JIS, BIG5 and GB2312 contain thousands of characters, and have a *variable-length* encoding scheme, while the terminals at the time were designed for single-byte character sets. In addition, glyphs for those characters were generally larger than their Latin counterparts. They solved this latter problem by allowing the terminal to combine two adjacent cells together into a “wide cell” and putting all the “wide” characters in their character set into wide cells by putting each byte of the wide character’s code into each half of a wide cell. Such a solution created a false connection between the quantity of the bytes used for encoding and the character’s visual width in fixed pitch (*monospace*) fonts. This is one of the origins of the East Asian Width property defined in UAX #11.


C:¥DOC¥M120501.JTX
編集 設定 配置 装飾 書式 拡張 検索 補助 終了 ヘルプ
1頁 10/ 40行 半角 R：かな 置換 文書更新中

```
                    見→ 積→ 書→

                                     昭和６３年１２月５日
○○株式会社　御中

下記の通りお見積もり致しましたので、
何卒ご用命のほどお願い申し上げます。

                8→ 0→ ，→ 0→ 0→ 0→ 円→

   （A）工事費             ６０，０００円
   （B）諸経費             ２０，０００円
   ※詳細は別紙明細の通りです。
```

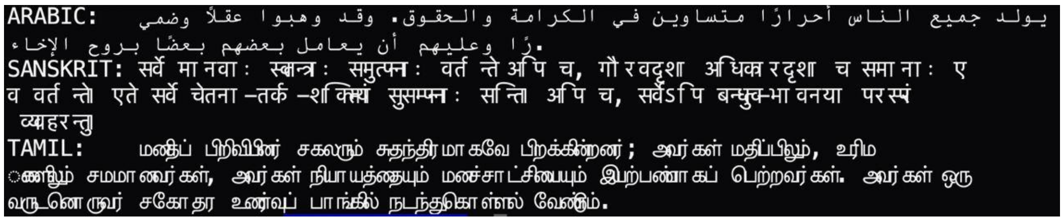
Figure 4 One screen of IBM 5550's Japanese application, showing wide characters. IBM 5550 is one of the widely adopted business computers in the late 1980s in Japan. It could be operated as an MS-DOS computer, a word processing machine, or a Japanese online terminal.

However, this mechanism could not be further extended to more complicated writing systems such as Arabic or Devanagari. In these writing systems, characters build up visuals in a much more complicated way: a grapheme may be a composite of multiple characters and one character’s visual, and width may change depending on its context. As a result, almost all existing terminals’ implementations produce broken results for these writing systems.



```
ARABIC:   يولد جميع الناس أحرارًا متساوين في الكرامة والحقوق. وق
د. وهبوا عقلاً وضميرًا وعليهم أن يعامل بعضهم بعضًا بروح الإخاء
SANSKRIT: स र्वे  मानवाः  स्वत  न्त्राः   समुत्प  न्नाः   व र्त  न्ते   अपि  च, ग
ौरवदृशा अधिकारदृशा च समानाः एव व र्त  न्ते । एते स र्वे  चेतना-तर्क -शक
्तिभ्यां  सुसम्प  न्नाः  स न्ति । अपि च, स र्वे ऽपि ब न्धु  त्व-भावनया परस्
परं  व्यवहर  न्तु ।
TAMIL:    மனிதப் பிறிவியினர் சகலரும் சுதந்திரமாகவே பிறக்கின்றனர்
; அவர்கள் மதிப்பிலும், உரிமைகளிலும் சமமானவர்கள், அவர்கள் நியாயத்
தையும் மனச்சாட்சியையும் இயற்பண்பாகப் பெற்றவர்கள். அவர்கள் ஒருவரு
டனொருவர் சகோதர உணர்வுப் பாங்கில் நடந்துகொள்ளல் வேண்டும்.
```

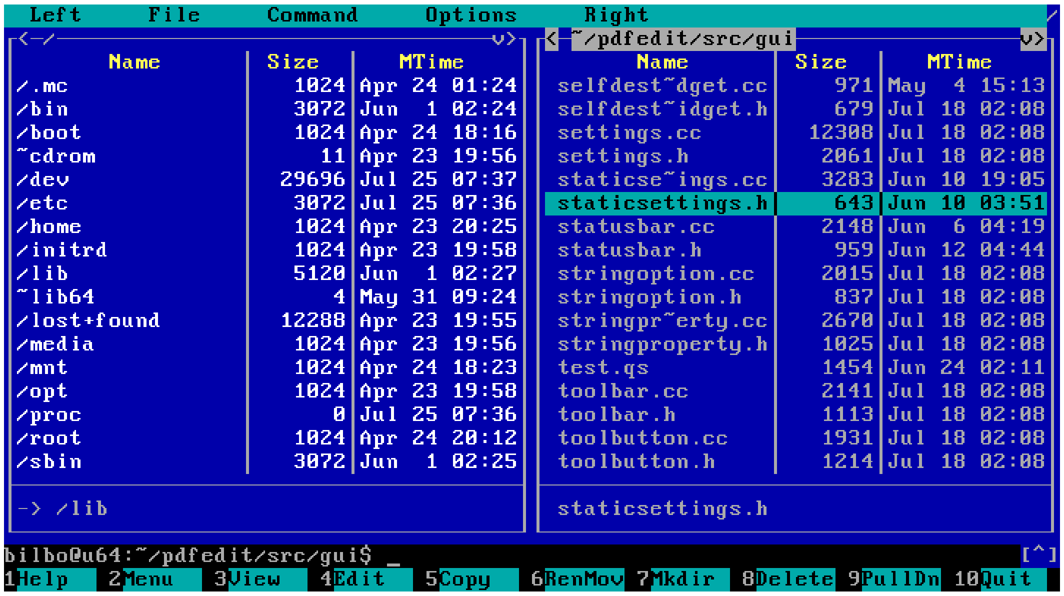
Figure 5 Display of Article I of Universal Declaration of Human Rights in Windows Terminal.



```
ARABIC:   يولد جميع الناس أحرارًا متساوين في الكرامة والحقوق. وقد وهبوا عقلاً وضمي
رًا. وعليهم أن يعامل بعضهم بعضًا بروح الإخاء.
SANSKRIT: सर्वे मानवाः स्वतन्त्राः समुत्पन्नाः वर्तन्ते अपि च, गौरवदृशा अधिकारदृशा च समानाः ए
व वर्तन्ते एते सर्वे चेतना-तर्क-शक्तिभ्यां सुसम्पन्नाः सन्ति अपि च, सर्वेऽपि बन्धुत्व-भावनया परस्पं
 व्यहरन्तु
TAMIL:    மனிதப் பிறிவியினர் சகலரும் சுதந்திரமாகவே பிறக்கின்றனர்; அவர்கள் மதிப்பிலும், உரிமை
களிலும் சமமானவர்கள், அவர்கள் நியாயத்தையும் மனச்சாட்சியையும் இயற்பண்பாகப் பெற்றவர்கள். அவர்கள் ஒரு
வருடனொருவர் சகோதர உணர்வுப் பாங்கில் நடந்துகொள்ளல் வேண்டும்.
```

Figure 6 Display of Article I of Universal Declaration of Human Rights in Apple's "Terminal.app".

In addition, many modern CLI and TUI applications provide "rich" terminal displays based on "drawing" characters (such as those defined in Unicode's Box Drawing and Block Elements blocks) to provide a more pleasant or navigable user interface. These applications usually analyze the text they are going to handle, perform some internal layout processing, and send complicated control sequences—such as cursor movement and style changes—to the terminal to produce these rich text user interfaces. As a result, these applications must be able to predict their hosting terminals' text layout behavior so they can determine the correct controls sent to the terminal.

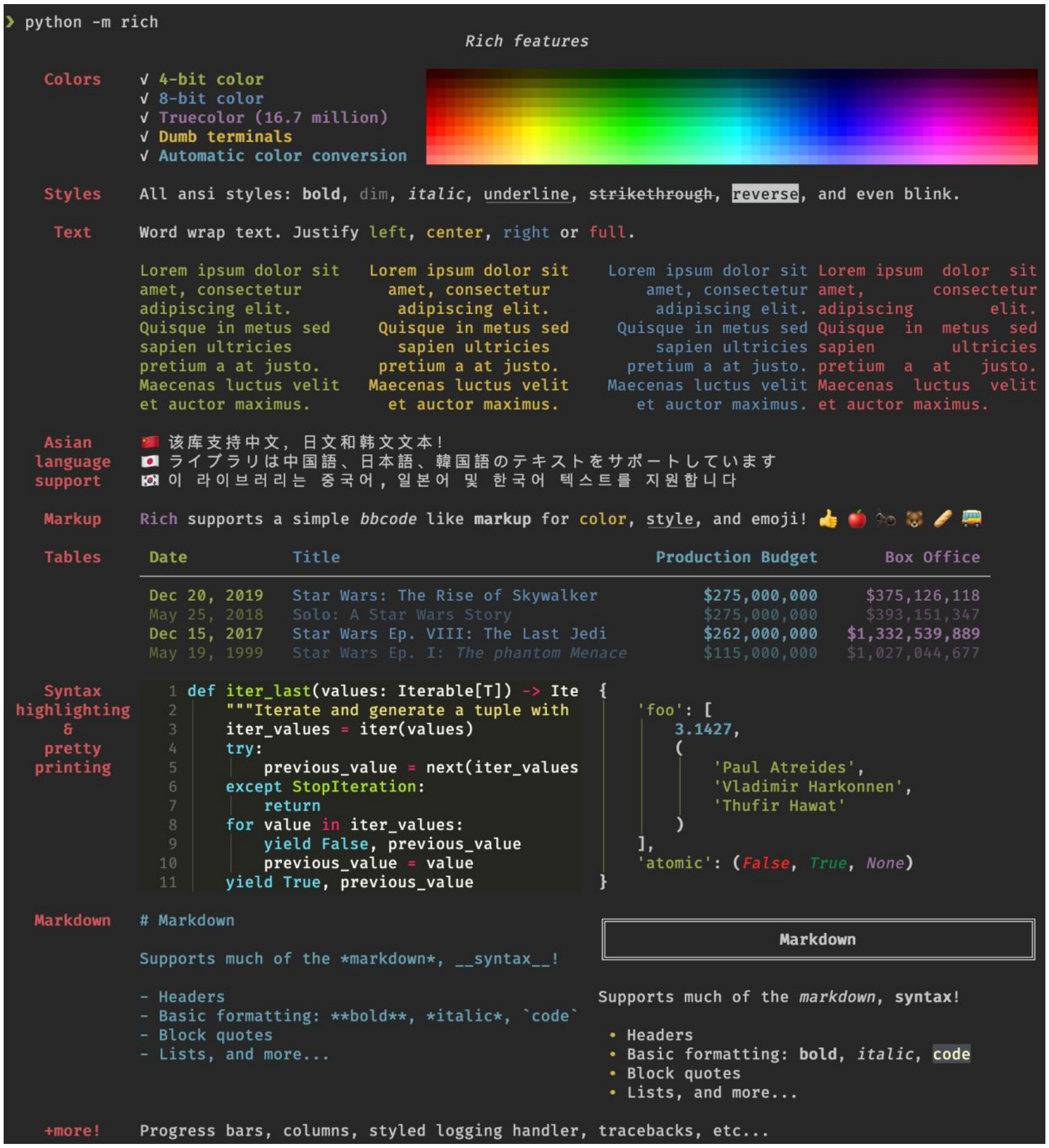


```
  Left     File     Command     Options     Right
┌<─~>──────────────────────────v>┐┌<~/pdfedit/src/gui─────────────v>┐
│     Name          │ Size │  MTime     ││    Name       │ Size │  MTime     │
│/.mc               │  1024│Apr 24 01:24││selfdest~dget.cc│   971│May  4 15:13│
│/bin               │  3072│Jun  1 02:24││selfdest~idget.h│   679│Jul 18 02:08│
│/boot              │  1024│Apr 24 18:16││settings.cc     │ 12308│Jul 18 02:08│
│~cdrom             │    11│Apr 23 19:56││settings.h      │  2061│Jul 18 02:08│
│/dev               │ 29696│Jul 25 07:37││staticse~ings.cc│  3283│Jun 10 19:05│
│/etc               │  3072│Jul 25 07:36││staticsettings.h│   643│Jun 10 03:51│
│/home              │  1024│Apr 23 20:25││statusbar.cc    │  2148│Jun  6 04:19│
│/initrd            │  1024│Apr 23 19:58││statusbar.h     │   959│Jun 12 04:44│
│/lib               │  5120│Jun  1 02:27││stringoption.cc │  2015│Jul 18 02:08│
│~lib64             │     4│May 31 09:24││stringoption.h  │   837│Jul 18 02:08│
│/lost+found        │ 12288│Apr 23 19:55││stringpr~erty.cc│  2670│Jul 18 02:08│
│/media             │  1024│Apr 23 19:56││stringproperty.h│  1025│Jul 18 02:08│
│/mnt               │  1024│Apr 24 18:23││test.qs         │  1454│Jun 24 02:11│
│/opt               │  1024│Apr 23 19:58││toolbar.cc      │  2141│Jul 18 02:08│
│/proc              │     0│Jul 25 07:36││toolbar.h       │  1113│Jul 18 02:08│
│/root              │  1024│Apr 24 20:12││toolbutton.cc   │  1931│Jul 18 02:08│
│/sbin              │  3072│Jun  1 02:25││toolbutton.h    │  1214│Jul 18 02:08│
│                                       ││                                    │
│-> /lib                                ││staticsettings.h                    │
└───────────────────────────────────────┘└────────────────────────────────────┘
bilbo@u64:~/pdfedit/src/gui$ _                                          [^]
1Help  2Menu  3View  4Edit  5Copy  6RenMov 7Mkdir  8Delete 9PullDn 10Quit
```

Figure 7 Example of a TUI application using Box Drawing characters and text styling

However, as mentioned above, applications know little about the terminal's text layout parameters and overall user-specified configuration. Therefore, there is no way apart from guessing—or a shared standard—for an application to determine how complex writing systems' text is going to be displayed, or how much space a string is going to occupy.

As a result, we find that a standardized specification for how to handle text in terminals is important work for the modern community of developers, and all other people that use text terminals.



```
❯ python -m rich
                                     Rich features

   Colors     √ 4-bit color
              √ 8-bit color
              √ Truecolor (16.7 million)
              √ Dumb terminals
              √ Automatic color conversion

   Styles     All ansi styles: bold, dim, italic, underline, strikethrough, reverse, and even blink.

    Text      Word wrap text. Justify left, center, right or full.

              Lorem ipsum dolor sit   Lorem ipsum dolor sit    Lorem ipsum dolor sit Lorem ipsum  dolor  sit
              amet, consectetur         amet, consectetur        amet, consectetur amet,       consectetur
              adipiscing elit.           adipiscing elit.         adipiscing elit. adipiscing        elit.
              Quisque in metus sed    Quisque in metus sed      Quisque in metus sed Quisque  in  metus  sed
              sapien ultricies           sapien ultricies         sapien ultricies sapien        ultricies
              pretium a at justo.     pretium a at justo.        pretium a at justo. pretium  a  at   justo.
              Maecenas luctus velit   Maecenas luctus velit    Maecenas luctus velit Maecenas  luctus  velit
              et auctor maximus.        et auctor maximus.        et auctor maximus. et auctor maximus.

   Asian      🇨🇳 该库支持中文，日文和韩文文本！
  language    🇯🇵 ライブラリは中国語、日本語、韓国語のテキストをサポートしています
  support     🇰🇷 이 라이브러리는 중국어, 일본어 및 한국어 텍스트를 지원합니다

   Markup     Rich supports a simple bbcode like markup for color, style, and emoji! 👍 🍎 🐜 🐻 🥖 🚌

   Tables      Date            Title                                 Production Budget      Box Office
              ─────────────────────────────────────────────────────────────────────────────────────────
               Dec 20, 2019    Star Wars: The Rise of Skywalker           $275,000,000    $375,126,118
               May 25, 2018    Solo: A Star Wars Story                    $275,000,000    $393,151,347
               Dec 15, 2017    Star Wars Ep. VIII: The Last Jedi          $262,000,000  $1,332,539,889
               May 19, 1999    Star Wars Ep. I: The phantom Menace        $115,000,000  $1,027,044,677

   Syntax     def iter_last(values: Iterable[T]) -> Ite  {
highlighting      """Iterate and generate a tuple with       'foo': [
     &            iter_values = iter(values)                     3.1427,
   pretty         try:                                           (
  printing            previous_value = next(iter_values              'Paul Atreides',
                  except StopIteration:                              'Vladimir Harkonnen',
                      return                                         'Thufir Hawat'
                  for value in iter_values:                      )
                      yield False, previous_value            ],
                      previous_value = value                 'atomic': (False, True, None)
                  yield True, previous_value                 }

  Markdown    # Markdown                                   ┌──────────────────────────────────────────┐
                                                           │                 Markdown                 │
              Supports much of the *markdown*, __syntax__! └──────────────────────────────────────────┘

              - Headers                                    Supports much of the markdown, syntax!
              - Basic formatting: **bold**, *italic*, `code`
              - Block quotes                                • Headers
              - Lists, and more...                          • Basic formatting: bold, italic, code
                                                            • Block quotes
                                                            • Lists, and more...

   +more!     Progress bars, columns, styled logging handler, tracebacks, etc...
```

Figure 8 Sample of Python library "Rich", showing its ability to do send complex controls to the terminal to produce a rich TUI.

Key Requirements

Normally when we do text layout in graphical user interface applications, aside from knowing the actual text being processed, we need to know its *style* (its font, its size, and other additional information). This applies to almost all existing text layout applications, like word processors, web browsers, etc. But this approach doesn't work for terminals since, as discussed in the previous section, *both the hosting terminal and guest application need to be able to do text layout, and the guest application does not fully know the text style being used in the terminal*.

Therefore, if we are going to create a TCSS specification, its layout algorithm can only rely on the text string being processed. Instead of taking fonts as *input*, the specification will define a set of requirements of fonts. Fonts that meet these requirements, referred to as "terminal fonts", will work properly with the layout algorithm defined in the specification. The results of layout, such as positions and metrics of the characters in the text, will be measured in terminal rows and cells.

As a result, a TCSS needs to contain the following:

- A specification for terminal *behaviors*, including:
    - Text writing behavior.
    - Cursor motion.
    - Selection and hit testing.
- A specification for terminal *text layout*, including:
    - Segmentation (clustering).
    - Measurement.
    - BiDi.
- Requirements for "terminal fonts".
- Additional control sequences, if required.

Text Writing Behavior

When a terminal receives text from its client applications, that text needs to be written into its *screen buffer*. The screen buffer will contain multiple *lines*, where one line corresponds to one *row* in the final visual screen. The terminal will also keep a *cursor*, representing the place that the text manipulation will happen.

Historically, one character is mapped into one visual cell, however, to support complex scripts, this constraint can no longer be satisfied. As a result, for terminals supporting complex scripts, each character in the screen buffer will have a metric measuring its *width*, denoting how many cells it will occupy. This is a non-negative integer, ranging from zero (for combining characters, etc.) to some relatively large number (e.g., for certain dependent vowels in Tamil) bounded only by the terminal emulator's available width.

In addition to *width*, characters may have other properties. For example, due to the requirements for BiDi reordering discussed in later sections, characters in the screen buffer will need to keep track of a *BiDi embedding level*, and the cursor will maintain both a *logical position* and *visual* *position*. Characters in the buffer will be ordered logically, since text editing manipulations are done in logical order.

Characters in one line will further be grouped into *terminal clusters*. A terminal cluster contains the characters that are combined together in the terminal environment. It is an instance of the *tailored grapheme cluster* defined in UAX #29. In Indic scripts, for example, syllables with virama conjoiners in the middle will be considered one single terminal cluster, while they are treated as multiple extended grapheme clusters in UAX #29.

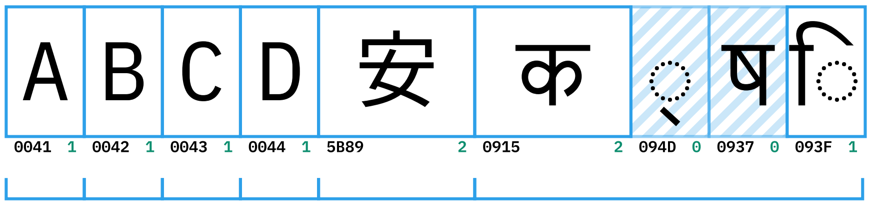


Figure 9 An example of one line in the screen buffer, showing characters, their width and the cluster boundaries. Characters measured in zero-width are marked with a striped background.

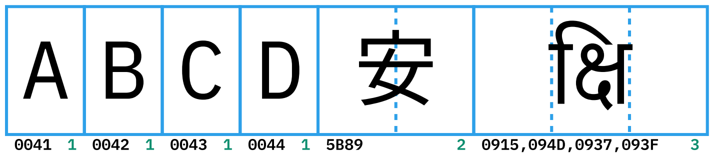


Figure 10 The screen buffer line shown visually. The 6th cluster (Devanagari क्षि) will occupy 3 cells but will contain 4 characters.

Text output in terminals typically works in overstrike mode: when writing, the terminal will always erase all cells occupied by any cluster that intersects any part of the incoming text, and then insert the new content. The erasing procedure will be done by replacing the existing clusters that occupy the needed space with blank clusters.

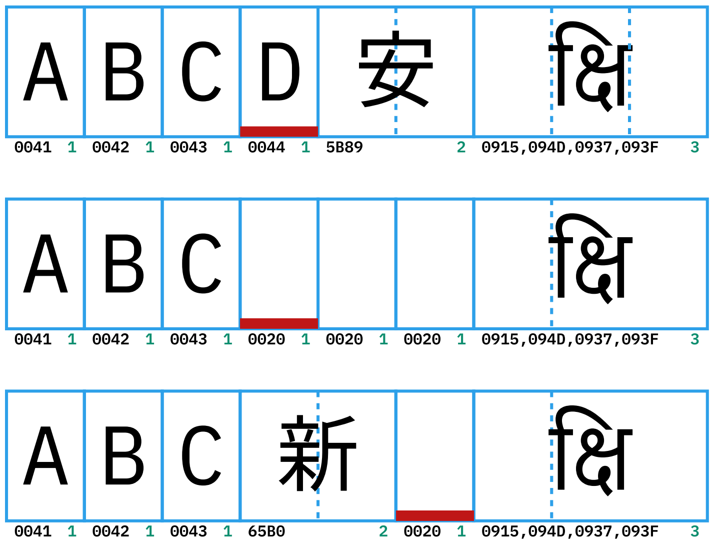


Figure 11 Example of writing U+65B0 over cell 4. In the erasing stage clusters 4 and 5 will be cleared into blanks, then the new content is overwritten.

In some writing systems, the form of a character may depend on the characters that follow it. One example of this is Devanagari's *repha* forms. This requires the establishment of a *work zone* that contains the most recent characters, and the property of the characters in the work zone is considered volatile and may change depending on the incoming text from the guest.

When the terminal receives text, it will first append the text into the work zone and measure the entire work zone to process potential property changes. If the measurement result says that the text in the work zone could be broken into multiple clusters, then the work zone will be shrunk to only contain the last (maybe incomplete) cluster. The text before that will be *committed*, and its properties will no longer change. As a result, at any time the work zone will contain at most **one cluster**. When the cursor moves (via the terminal receiving a cursor move command or a newline), all the text in the work zone will be committed—even if it is incomplete—and the work zone will be cleared.

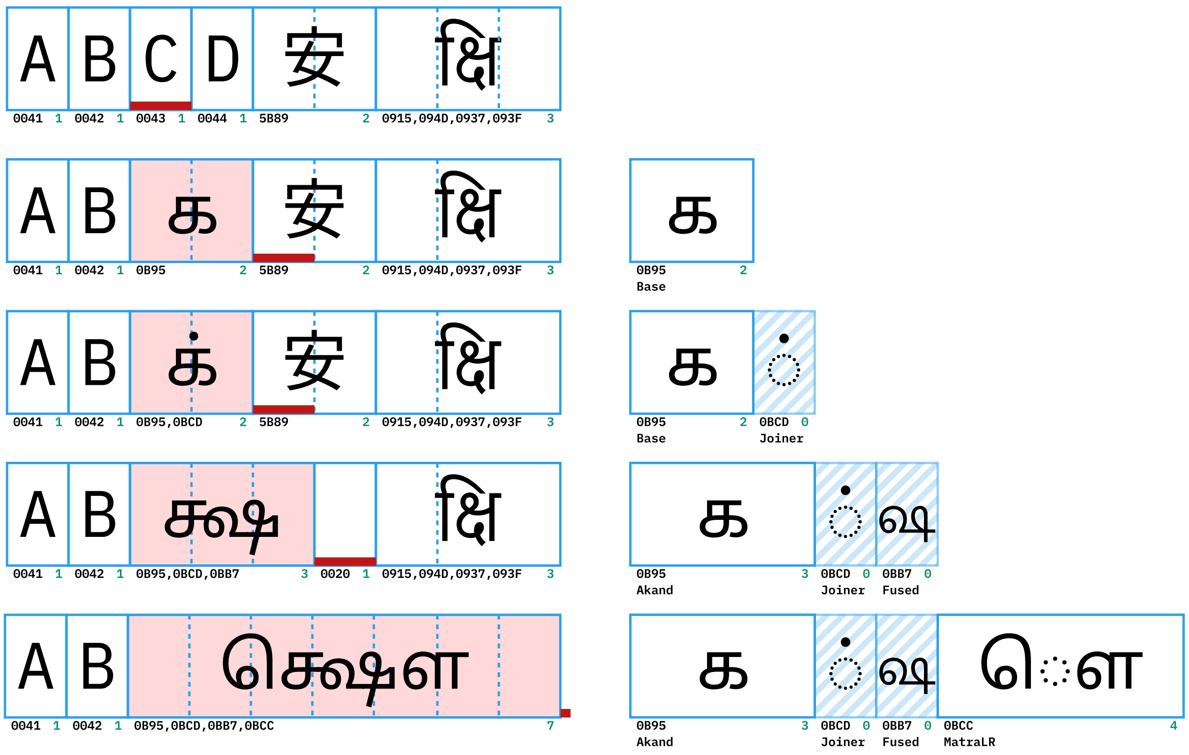

A B C D 安 क्षि
0041 1 0042 1 0043 1 0044 1 5B89 2 0915,094D,0937,093F 3
A B க 安 क्षि
0041 1 0042 1 0B95 2 5B89 2 0915,094D,0937,093F 3
க
0B95 2 Base
A B க் 安 क्षि
0041 1 0042 1 0B95,0BCD 2 5B89 2 0915,094D,0937,093F 3
க ்
0B95 2 Base 0BCD 0 Joiner
A B க்ஷ क्षि
0041 1 0042 1 0B95,0BCD,0BB7 3 0020 1 0915,094D,0937,093F 3
க ் ஷ
0B95 3 Akand 0BCD 0 Joiner 0BB7 0 Fused
A B க்ஷௌ
0041 1 0042 1 0B95,0BCD,0BB7,0BCC 7
க ் ஷ ௌ
0B95 3 Akand 0BCD 0 Joiner 0BB7 0 Fused 0BCC 4 MatraLR

Figure 12 The process of writing U+0B95, 0BCD, 0BB7, 0BCC. The dark red zone is the work zone. The right column shows the form and width of each character in the work zone. After U+0BB7 is written, the width of U+0B95 is set to 3, and its form is set to "Akhand". As a result, the work zone expands and overstrikes the U+5B89 cluster after it.

Soft line breaks in the terminal are done at cluster boundaries, instead of following UAX #14. Considering that the work zone only contains the last cluster, line breaking in a terminal could be summarized as follows: when the work zone grows too wide to be contained within the current line, it will be moved to the beginning of the next line, and the cells it used to occupy will be erased by spaces.

Measurement

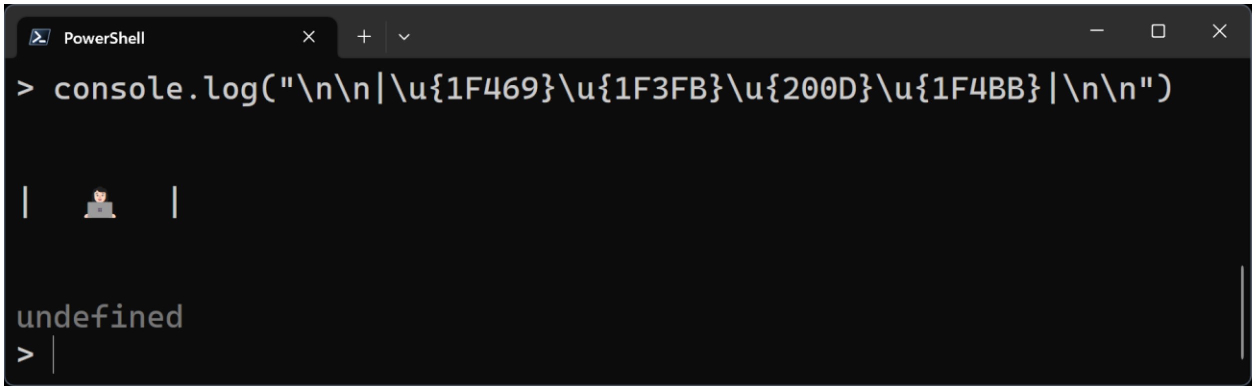
The core part of a TCSS is *measurement*. A terminal must know the space a text occupies before it can allocate cells in its screen buffer. As shown in previous sections, many existing terminal implementations produce broken results when dealing with complex scripts such as Devanagari and Tamil. The glyphs are placed at incorrect places, showing that the existing implementations' current means of measuring text is not compatible with those complex scripts.

At status quo, most terminals and their guest applications use very simple methods to measure the text. Most of them are derived from UAX #11 East Asian width, extended to support more characters. Terminal implementations on Unix-like systems usually use a POSIX function called *wcwidth* to query the space that a Unicode character occupies. However, this function only takes one argument—which is a single codepoint—and returns a fixed value representing its width, which is clearly insufficient for complex scripts where multiple codepoints contribute to each grapheme cluster.

For those complex scripts, a character's form varies depending on its surrounding context. Therefore, its width varies as well. One well-known example manifesting this issue is Emoji: if an Emoji character is joined with a preceding emoji character in a Zero-Width Joiner (ZWJ) emoji sequence, that character will no longer occupy any space since it is now "fused" with the character before it (Table 1). If the terminal uses *wcwidth* to measure the space a composite Emoji occupies, it will allocate *too much* space for the Emoji, leaving a lot of space around the actual text (see Figure 13).

Table 1 A small collection of text, wcwidth results and "expected" results.

👩🏻‍💻	**Character**	**U+1F469**	**U+1F3FB**	**U+200D**	**U+1F4BB**
	wcwidth result	2	2	0	2
	Expected width	2	0	0	0
	Expected form	"Base"	"Modifier"	"Joiner"	"Fused"
क्षि	**Character**	**U+0915**	**U+094D**	**U+0937**	**U+093F**
	wcwidth result	1	0	1	1
	Expected width	2	0	0	1
	Expected form	"Akhand"	"Joiner"	"Fused"	"MatraL"
க்ஷௌ	**Character**	**U+0B95**	**U+0BCD**	**U+0BB7**	**U+0BCC**
	wcwidth result	1	0	1	1
	Expected width	3	0	0	4
	Expected form	"Akhand"	"Joiner"	"Fused"	"MatraLR"
ರ್ಝೈ	**Character**	**U+0CB0**	**U+0CCD**	**U+0C9D**	**U+0CC8**
	wcwidth result	1	0	1	1
	Expected width	2	0	3	1
	Expected form	"Repha"	"Joiner"	"Base"	"MatraUR"
ರ಼್ಚ	**Character**	**U+0CB0**	**U+0CBC**	**U+0CCD**	**U+0C9A**
	wcwidth result	1	0	0	1
	Expected width	2	0	0	1
	Expected form	"Base"	"Nukta"	"Joiner"	"Subjoin"



```
> console.log("\n\n|\u{1F469}\u{1F3FB}\u{200D}\u{1F4BB}|\n\n")


|  👩🏻‍💻  |


undefined
>
```

Figure 13 Windows Terminal showing an Emoji with skin tone modifier and ZWJ components. Due to incorrect measurement, extra spaces are added around the Emoji.

Therefore, a proper measurer has the following requirements:

1. It must take a *string* as input to fully understand the context.
2. It must be able to break down the string into *terminal clusters*.
3. It must determine the *form* for every character in the input.
4. It must determine the *width* for every character in the input.

Objective 2 suggests that the measurement algorithm should have a *cluster model* that applies for most of the writing systems. As a result, we propose the creation of a cluster model that is simplified from the cluster model of Andrew Glass' Universal Shaping Engine. The basic cluster model applies as follows:

$$\begin{aligned}\text{Cluster}_{\mathscr{s}} &= \text{PreControl}_{\mathscr{s}}^{*}\ \text{Pre}_{\mathscr{s}}^{*}\ (\text{Base}_{\mathscr{s}}\ \text{Post}_{\mathscr{s}}^{*})\ (\text{Joiner}_{\mathscr{s}}^{+}\ \text{Base}_{\mathscr{s}}\ \text{Post}_{\mathscr{s}}^{*})^{*}\ \text{PostControl}_{\mathscr{s}}^{*}\\ \text{Cluster} &= \bigcup_{\mathscr{s}} \text{Cluster}_{\mathscr{s}}\end{aligned}$$

For each script $\mathscr{s}$, there will be a collection of character sets:

- $\text{PreControl}_{\mathscr{s}}$: the pre-cluster control characters.
- $\text{PostControl}_{\mathscr{s}}$: the post-cluster control characters.
- $\text{Base}_{\mathscr{s}}$: the base characters.
- $\text{Pre}_{\mathscr{s}}$: the pre-base additional characters, like the pre-consonant modifiers in Thai.
- $\text{Post}_{\mathscr{s}}$: the post-base additional characters, like diacritics.
- $\text{Joiner}_{\mathscr{s}}$: the joiners that combines multiple bases, like Virama or ZWJ.

The character sets $\text{Base}_{\mathscr{s}}$ for different scripts will be mutually exclusive, while the other character sets may have common characters across different scripts (for example, ZWJ may be used as joiner among many scripts). Since the definition of "Cluster" is a regular language, the clustering procedure could be efficiently implemented using a finite state machine.

To accomplish objective 3, the algorithm needs to assign a "form" information for each character in the string. Considering the complexity of the writing systems that Unicode supports, the form decision process will need to be rule-driven, i.e., there will be one single *algorithm*, and a set of rules that is used as the knowledge database for the algorithm. We could have the form-decision rules for Latin, Devanagari, Arabic and Emoji, then they combined will be used as the rule set of the algorithm.

The form assignment process could be implemented as the "side effects" of the state machine that does the clustering: when the state machine identifies additional parts of the cluster, we will be able to use the current state and the incoming character to determine the form of the incoming character and change the forms of the existing characters in the cluster if necessary.

As a result, the measurement part of the specification will contain the following components:

- Measurement algorithms, mostly the form analyzer and the segmenter.
- The measurement algorithm's rule set.
- Database for character widths, using code point and form as query keys.

Considering the wide adoption of *wcwidth*, the proposed measuring algorithms should return the same results as *wcwidth* if the input is considered to contain only "simple" characters.

BiDi and Cursor Movement

Supporting bidirectional text (BiDi) is another important feature for TCSS. To support BiDi, the terminal's screen buffer will need to retain the following information:

- Boundaries of paragraphs, since BiDi embedding level analysis is done per-paragraph.
- Embedding level, which is recalculated every time the text inside a paragraph is changed.
- Potential special controlling data (such as an embedding level override) which will be useful for complex TUI applications that do the text layout on their own, e.g., full-screen text editors and IDEs.

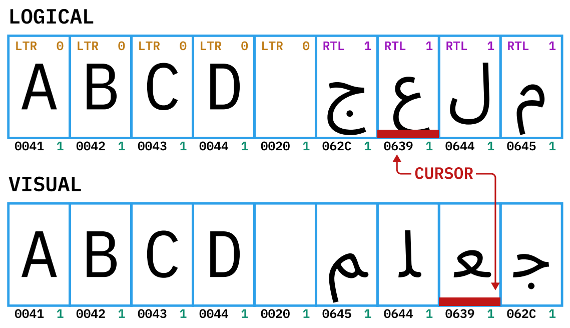

LOGICAL
LTR 0 LTR 0 LTR 0 LTR 0 LTR 0 RTL 1 RTL 1 RTL 1 RTL 1
A B C D ج ع ل م
0041 1 0042 1 0043 1 0044 1 0020 1 062C 1 0639 1 0644 1 0645 1
CURSOR
VISUAL
A B C D م ل ع ج
0041 1 0042 1 0043 1 0044 1 0020 1 0645 1 0644 1 0639 1 062C 1

Figure 14 Logical buffer and its visual representation of a line containing both Latin and Arabic. The embedding level is shown in the logical buffer row. After reordering, the order of characters in the visual representation is no longer identical to that in the logical buffer. As a result of reordering, the cursor will have both a logical position and a visual position.

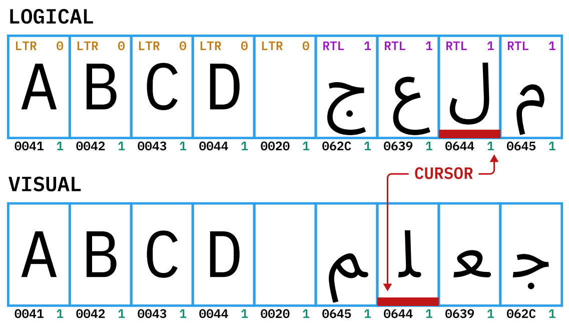

LOGICAL
LTR 0 LTR 0 LTR 0 LTR 0 LTR 0 RTL 1 RTL 1 RTL 1 RTL 1
A B C D ج ع ل م
0041 1 0042 1 0043 1 0044 1 0020 1 062C 1 0639 1 0644 1 0645 1
CURSOR
VISUAL
A B C D م ل ع ج
0041 1 0042 1 0043 1 0044 1 0020 1 0645 1 0644 1 0639 1 062C 1

Figure 15 The cursor after moving left by one cell. Logically, it moves to the end of the line.

Every time text is written into the terminal, UAX #9 embedding level analysis is run on the paragraphs being modified, and the resulting embedding levels will be used for the final *reordering*. Reordering is done line-by-line: the text in the same line will be reordered according to its embedding level, and the result will be sent to the display.

As a result of reordering, the cursor will have a *logical position* and a *visual position*. While doing text writing is mainly using the logical position, the cursor movement operates at visual level, i.e., when the terminal decides to move the cursor *upward*, it will need to look up a logical position in the previous line that has the same column in its visual counterpart.

Selection and Hit Testing

In modern terminals, selection could be done in two ways: either to select a range of the text in the logical order or select a *rectangle* of the text in visual order. To perform all these selections, as well as the cursor movement, the terminal must be able to perform *hit testing*, which is converting visual positions (row, column) into logical positions (line, character, *cell*).

We would define two procedures, "find the first character that occupies a certain cell" and "find the last character that occupies a certain cell", which takes the visual position and a screen buffer line, and then find the first or last character in the line that occupies a cell. Using these procedures, terminals could implement selection.

Font Requirements

Since in TCSS all the text measurement is done *without* the font, to produce a correct final display, the font used in the terminal might need to also meet certain conditions, making the glyphs' metrics or shaping behavior match TCSS' expectations. The TCSS project could define a series of typographic features so the fonts could be applied with those features enabled to produce the correct results.

Control Sequences

In some cases, we may need a series of control sequences that client applications could send to the terminal for special purposes, including:

- Overriding specific text metrics.
- Overriding the default BiDi behavior or information, such as the embedding level.
- Breaking a BiDi paragraph in the middle of a line without introducing a carriage return or line feed.

The TCSS project should figure out the places that special controls may be present and design the control sequences that could be discussed in other standardization organizations, like ECMA or ISO.

[End of document]