

# Rich Text Dialect

## Table of Contents

1. Overview .....	1
2. High-level Rich-Text Dialect (RTD) .....	1
3. Low-level Styling Dialect .....	3
4. Rich text face type .....	4
4.1. Single-box mode .....	4
4.2. Multi-box mode .....	5
5. Info querying functions .....	5

## 1. Overview

Rich-Text is text that supports formatting. Different graphic styles can be applied to text or to segments of text.

The Rich-Text API has three different levels, from simplest to most optimized:

- Using RTD (from VID, or when manually constructing the face).
- A low-level rich styling dialect for one text string (for fast performance).
- Multiple rich text paragraphs in a single face (for complex layouts).

## 2. High-level Rich-Text Dialect (RTD)

### Grammar

```

nested: [ahead block! into rtd]
color: [
  s: tuple! (v: s/1) ;-- color as R.G.B tuple
  | issue! (v: hex-to-rgb s/1) ;-- color as #rgb or #rrggbb hex
value
  | word! if (tuple? attempt [v: get s/1])
]
f-args: [
  ahead block! into [integer! string! | string! integer!]
  | integer!
  | string!
]
style!: make typeset! [word! tag! tuple! path!]
style: [ahead style! [
  ['b | 'bold | <b>] (push 'b) [nested | rtd [/b | /bold | </b>]] (pop
'b)
  | ['i | 'italic | <i>] (push 'i) [nested | rtd [/i | /italic | </i>]] (pop
'i)
  | ['u | 'underline | <u>] (push 'u) [nested | rtd [/u | /underline | </u>]] (pop
'u)
  | ['s | 'strike | <s>] (push 's) [nested | rtd [/s | /strike | </s>]] (pop
's)
  | ['f | 'font | <font>]
    s: f-args (push either block? s/1 [head insert copy s/1 'f][reduce ['f s/1]])
    [nested | rtd [/f | /font | </font>]]
    (pop 'f)
  | ['bg | <bg>] color (push reduce ['bg v]) [nested | rtd [/bg | </bg>]] (pop 'bg)
  | color (push-color v) opt [nested (pop-color)]
  | ahead path!
  into [
    (col: 0 insert/only mark tail stack) some [ ;@@ implement any-
single
    (v: none)
    s: ['b | 'i | 'u | 's | word! if (tuple? attempt [v: get s/1])]
    (either v [col: col + 1 push-color v][push s/1])
    ](insert cols col)
  ]
  nested (pop-all take mark)
]]
rtd: [some [pos: style | s: [string! | char!] (append text s/1 s-idx: tail-idx?)]]

```

## NOTE

The path syntax allows to combine several styles together to be applied on the block following the path. Mixed usage of delimiters and blocks for different styles is allowed.

## Usage

RTD input is processed by a specific **rtd-layout** function that will return a single-box rich-text face, where the RTD code will be compiled to a single text string (stored in /text facet) and a low-level

styling description (stored in /data facet).

The full specification of the function is:

```
rtd-layout func [  
    "Returns a rich-text face from a RTD source code"  
    spec [block!] "RTD source code"  
    /only "Returns only [text data] facets"  
    /with "Populate an existing face object"  
        face [object!] "Face object to populate"  
    return: [object! block!]  
]
```

Example:

```
rt: rtd-layout [<i> <b> "Hello" </b> <font> 24 red " Red " </font> blue "World!" </i>]  
  
view [rich-text with [text: rt/text data: rt/data]]
```

RTD may be directly provided in VID for the rich-text **data** facet.

Examples:

```
view compose [rich-text 200x100 data [i b "Hello" /b font 24 red " Red " /font blue  
"World!" /i]]  
  
view compose [rich-text 200x100 data [i [b ["Hello"]] red font 24 [" Red "] blue  
"World!"]]]  
  
view compose [rich-text 200x100 data [i/b/u/red ["Hello" font 32 " Red " /font blue  
"World!"]]]  
  
view compose [rich-text 200x100 data [i/blue ["Hello " b/u/red [font [32 "Arial"] "Red  
" /font] "World!"]]]
```

### 3. Low-level Styling Dialect

This dialect describes a list of styles to be applied on the string referred by /text facet in a rich-text face. The purpose of this dialect is to provide a solution for dynamic changes and info querying that performs as fast as possible. This also maps well with the underlying hardware-accelerated APIs (it relies on Direct2D on Windows).

#### Usage

The dialect grammar is a simple list of text segments (defined using a starting position and a length combined in a pair! value) followed by a list of styles. So, the typical structure is:

```
[
  <range1> style1 style2 ...      ;-- range1: start1 x length1
  <range2> style1 style2 ...      ;-- range2: start2 x length2
  ...
]
```

Styles can overlap, and later styles have higher priority (cascading styles).

The following styles are supported:

```
[
  tuple!                                ;-- text color
  | backdrop tuple!                     ;-- background color
  | bold                                ;-- bold font
  | italic
  | underline tuple! (color) lit-word! ('dash, 'double, 'triple) ;@@ color and type
are not supported yet
  | strike tuple! (color) lit-word! ('wave, 'double)           ;@@ color and type
are not supported yet
  | border tuple! (color) lit-word! ('dash, 'wave)             ;@@ not
implemented
  | integer!                                ;-- new font size
  | string!                                ;-- new font name
]
```

#### NOTE

Text's color should not follow immediately after **strike** or **underline**. Color and type for **strike** and **underline** will modify their line styles, not text. As they are not yet implemented, specifying color (or type) after these keywords will have no effect.

## 4. Rich text face type

A new native rich-text face type supports rich text features with underlying hardware-acceleration. The face has two modes for displaying rich text.

### 4.1. Single-box mode

The whole face area is used for displaying the rich text, starting at upper left corner, using the following specific facets:

- `/data (block!)`: a block of low-level styling dialect instructions to be applied on text facet.
- `/text (string!)`: a text string to be displayed using the `/data` facet styles description.

Draw facet can still be used and it will be rendered on top of the rich text display.

Examples:

```
view [
  rich-text with [
    text: "Hello Red World!"
    data: [1x17 0.0.255 italic 7x3 255.0.0 bold 24 underline]
  ]
]
view [
  rich-text "Hello Red World!"
  with [data: [1x17 0.0.255 italic 7x3 255.0.0 bold 24 underline]]
]
```

## 4.2. Multi-box mode

In this mode, an arbitrary number of rich text areas can be displayed inside the same rich-text face. In order to achieve that, each rich text area is specified using the text keyword in Draw dialect.

Specific facets:

- `/draw (block!)`: a block of text instructions, eventually mixed with regular Draw instructions.
- `/text (none!)`: this facet must be set to `none` in order to enable this mode.

### Draw extension

```
text <pos> <text>
```

`<pos>` : a pair! value indicating the upper left corner of the text-box.

`<text>` : a string, or a rich-text face object with a rich-text description in single-box

Example:

```
view compose/deep [
  rich-text 200x200 draw [
    text 10x10 (rt1: rtd-layout ["Some^/" b "text^/" /b "here"] rt1/size: 50x80
    rt1)
    text 100x90 (rt2: rtd-layout [red "Other^/" b "text^/" /b "there"] rt1/size:
    50x80 rt2)
    pen gold box 90x80 160x180
  ]
]
```

## 5. Info querying functions

The following functions are provided to query information about the content of a rich-text face. These functions can be used to easily implement:

- cursor navigation
- hit testing

From the default context **system/words**:

```
caret-to-offset: function [
  "Given a text position, returns the corresponding coordinate relative to the top-
  left of the layout box"
  face    [object!]
  pos     [integer!]
  return: [pair!]
]

offset-to-caret: function [
  "Given a coordinate, returns the corresponding text position"
  face    [object!]
  pt      [pair!]
  return: [integer!]
]

size-text: function [
  "Returns the area size of the text in a face"
  face [object!]
  /with                                ;-- unused for rich-text
    text [string!]
  return: [pair! none!]
]
```

From the rich-text context:

```
line-height?: function [
  "Given a text position, returns the corresponding line's height"
  face    [object!]
  pos     [integer!]
  return: [integer!]
]

line-count?: function [
  "number of lines (> 1 if line wrapped)"
  face    [object!]
  return: [integer!]
]
```

Examples:

```

view [
  rich-text data [font 16 "Select some text with your mouse" /font]
  on-down [
    bkg: reduce [ ; Background for selected text
      as-pair caret: offset-to-caret face event/offset 0
      'backdrop sky
    ]
    either 2 = length? face/data [ ; On first selection
      pos: tail face/data
      append face/data bkg
    ][ ; Changing starting pos on subsequent selections
      change pos bkg/1
    ]
  ] all-over
  on-over [
    if event/down? [ ; On dragging change only length
      pos/1/2: (offset-to-caret face event/offset) - caret
    ]
  ]
]

```

```

view compose/deep [
  rich-text draw [
    text 10x10 (rt: rtd-layout [i/blue ["Hello " red/b [font 24 "Red " /font]
    "World!"]])
    line-width 5 pen gold
    line ; Let's draw line under words using a pair of above helper functions
      (as-pair 10 h: 10 + rich-text/line-height? rt 1) ; Starting-point y -> 10
    + line-height
      (as-pair 10 + pick size-text rt 1 h) ; End-point x -> 10 + length-of-text-
    size
  ]
]

```