

# Final Report

Fan Chung  
August 19, 2021

## Contents

<b>1</b>	<b>Introduction to vt(4)</b>	<b>1</b>
1.1	What is vt(4)?	1
1.2	vt(4) in Kernel	2
1.3	Digging into vt(4)	2
1.3.1	Input	2
1.3.2	Output	4
1.3.3	Terminal Features	4
<b>2</b>	<b>My works</b>	<b>4</b>
2.1	Introduction	5
2.2	Backend	5
2.3	Frontend	7
2.3.1	vt_ime_send_message	8
2.3.2	vt_ime_send_char	9
2.3.3	vt_ime_delete	9
2.3.4	vt_ime_request_output	9
2.3.5	vt_ime_check_valid_char	9
2.3.6	vt_ime_convert_utf8_byte	9
2.3.7	vt_ime_input	9
2.3.8	vt_ime_toggle_mode	10
2.3.9	vt_ime_is_enabled	10
2.3.10	vt_ime_process_char	10
2.3.11	vt_ime_draw_status_bar	10
<b>3</b>	<b>Conclusion</b>	<b>10</b>

## 1 Introduction to vt(4)

### 1.1 What is vt(4)?

According to man page vt(4), vt is a *virtual terminal console driver*, and it provides multiple virtual terminals with an extensive feature set.

For example:

- Unicode UTF-8 text with double-width characters.
- Large font maps in graphics mode, including support for Asian character sets.
- Graphics-mode consoles.
- Integration with KMS (Kernel Mode Setting) video drivers for switching between the X Window System and virtual terminals.

## 1.2 vt(4) in Kernel

The kernel sources of `vt` are located under `src/sys/dev/vt`. The `vt` component including the following main files:

- `hw/`: contain the frame buffer implementations for different hardware drivers, such as `vga`, `fb`.
- `logo/`: for storing cpu logos pixel data. (`logo_beastie.c` and `logo_freebsd.c`)
- `font/`: for storing the default font and mouse cursor pixel data.
- `color/`: for customizing color palette entries. (set with `kern.vt.color.<colnum>.rgb`)
- `vt.h`: the main header file, containing struct/function declarations of sources.
- `vt_core.c`: the main source file, containing instances of structures and APIs
- `vt_buf.c`: for accessing and manipulating console data buffer.
- `vt_font.c`: for loading fonts when the graphic mode is set.
- `vt_cpulogos.c`: for drawing cpu logos during booting.
- `vt_sysmouse.c`: for defining virtual mouse device driver `sysmouse(4)`
- `vt_consolectl.c`: ?

## 1.3 Digging into vt(4)

But you might ask the questions like:

- How to understand the so-called *console driver*?
- Is it a console or a driver?
- What are requirements for becoming a console driver?

Don't worry! In this section, we will explore the internal of `vt` and introduce what they do and how they work. Let's go!

To understand `vt` more easily, you can imagine `vt` is a regular terminal emulator application, for example, `xterm`. The only difference between them is `vt` lies in kernel, that is, its components you interact with work in a low-level way.

For any terminal emulator, you can type commands on it, and it will show you the results. Or you can move the mouse to copy/paste some text and scroll the mouse wheel to navigate between history. Moreover, you can use control sequences to move the cursor, changes screen colors and etc. `vt` has no exception. In general, the features mentioned above corresponds to three main components of `vt`: input, output and terminal features.

### 1.3.1 Input

1. Keyboard The keyboard part is
2. Mouse

With a mouse, one can copy and paste text from the screen in `vt`.

The mouse feature in `vt` is implemented with `sysmouse(4)`, a virtualized mouse driver.

Quoted from sysmouse(4)

The console driver, in conjunction with the mouse daemon moused(8), supplies mouse data to the user process in the standardized way via the sysmouse driver. This arrangement makes it possible for the console and the user process (such as the X Window System) to share the mouse.

The following code snippet shows the function `sysmouse_drvinit()` creates a device called `/dev/sysmouse`. Note that the `SYSINIT` in the last line do sysmouse driver initialization.

```
1  /* sys/dev/vt/vt_sysmouse.c:477 */
2  static void
3  sysmouse_drvinit(void *unused)
4  {
5
6      if (!vty_enabled(VTY_VT))
7          return;
8      mtx_init(&sysmouse_lock, "sysmouse", NULL, MTX_DEF);
9      cv_init(&sysmouse_sleep, "sysmrd");
10     make_dev(&sysmouse_cdevsw, 0, UID_ROOT, GID_WHEEL, 0600,
11             "sysmouse");
12     #ifdef EVDEV_SUPPORT
13         sysmouse_evdev_init();
14     #endif
15 }
16
17 SYSINIT(sysmouse, SI_SUB_DRIVERS, SI_ORDER_MIDDLE, sysmouse_drvinit, NULL);
```

The following code snippet shows the non-static function `sysmouse_process_event()` fires up `vt_mouse_event()` and pass the mouse information.

```
1  /* sys/dev/vt/vt_sysmouse.c:202 */
2  void
3  sysmouse_process_event(mouse_info_t *mi)
4  {
5      /* ... */
6
7      #ifndef SC_NO_CUTPASTE
8          mtx_unlock(&sysmouse_lock);
9          vt_mouse_event(mi->operation, x, y, mi->u.event.id, mi->u.event.value,
10                      sysmouse_level);
11      return;
12      #endif
```

And the function `sysmouse_process_event()` is invoked by `consolectl_ioctl()` in `vt_consolectl.c`. Everything seems so reasonable!

```
1  /* src/sys/dev/vt/vt_consolectl.c:50 */
2  static int
3  consolectl_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int flag,
4                  struct thread *td)
5  {
6      /* ... */
7
8      case CONS_MOUSECTL: {
9          mouse_info_t *mi = (mouse_info_t*)data;
10
11          sysmouse_process_event(mi);
12          return (0);
13      }
```

Finally, depending on different mouse actions and events, `vt_mouse_event()` will do corresponding behaviors on the screen such as marking, copying, pasting and etc. You may refer to `sys/dev/vt/vt_core.c:2136` for more details.

### 1.3.2 Output

vt provides several hardware backends

### 1.3.3 Terminal Features

```
1  /* sys/dev/vt/vt_core.c:90 */
2  const struct terminal_class vt_termclass = {
3      .tc_bell = vtterm_bell,
4      .tc_cursor = vtterm_cursor,
5      .tc_putchar = vtterm_putchar,
6      .tc_fill = vtterm_fill,
7      .tc_copy = vtterm_copy,
8      .tc_pre_input = vtterm_pre_input,
9      .tc_post_input = vtterm_post_input,
10     .tc_param = vtterm_param,
11     .tc_done = vtterm_done,
12
13     .tc_cnprobe = vtterm_cnprobe,
14     .tc_cngetc = vtterm_cngetc,
15
16     .tc_cngrab = vtterm_cngrab,
17     .tc_cnungrab = vtterm_cnungrab,
18
19     .tc_opened = vtterm_opened,
20     .tc_ioctl = vtterm_ioctl,
21     .tc_mmap = vtterm_mmap,
22 };
```

## 2 My works

## 2.1 Introduction

As the proposal stated, this project aims to provide an environment that can run IME (input method engine) to enable users to type CJK characters in `vt`.

This project was divided into two parts, backend and frontend. The backend is supposed to process keys sent from the frontend and translate them into valid CJK characters, depending on different input schemas. The frontend, on the other hand, receives utf-8 encoded CJK characters and insert them on the screen. Additionally, the frontend need to print preedit string and candidates during composing.

## 2.2 Backend

To facilitate the software development process, I choose *Python* as our backend development language. Compared with other programming languages, *Python* is renowned for its easy-to-use APIs as well as being an interpreted language.

I started by implementing FFI between C and Python with *ctypes* to access C APIs provided in *librime*. However, I found it's difficult to fully implement the mappings from *librime*'s structs and functions to *Python*'s own data types. As a result, I decided to add a C wrapper to define my custom data members and methods to encapsulate those *librime*'s APIs and compiled it into a shared library to be loaded with *ctypes*. Thus I can use Python to write the backend. The code of this part can be found in the directory `tmux_rime/rime_wrapper`<sup>1</sup> in previously listed `tmux-rime` repository.

The next step is to consider how to accomplish the communication between the frontend and backend. The frontend needs to receive keys from the user, then waits for the backend sending the results back. However, you never know when a user will finish composing. For example, if a user presses a single key '5' on a standard QWERTY keyboard, which is mapped to a *initial*<sup>2</sup> (聲母 in chinese) 「ㄅ」 (chih in *Wade-Giles* romanization) in the input schema *Bopomofo*<sup>3</sup>, the IME server can't decide whether there still have key sequences or not, since the user can continue to press the keys mapped to Four tones,<sup>4</sup> such as pressing the key '3' to compose 「ㄅ ˊ」 or the space key to compose 「ㄅ」 with *even* (平 píng) tone. Therefore, the backend IME server is required to keep running in background for listening requests from the frontend. I wrote server-side code with Python's `socketserver`. You may refer to `tmux_rime/rime_wrapper/tmux_rime_server.py`<sup>5</sup>

Before implementing the frontend in `vt`, I implemented the frontend for `tmux` called `tmux-rime` for the GSoC first stage evaluation as a proof of concept. It also serves as a test ground for the backend and *librime*. The following figures shows the structure and the screenshot of `tmux-rime`.

---

<sup>1</sup>[https://gitlab.com/Cycatz/tmux-rime/-/tree/main/tmux\\_rime](https://gitlab.com/Cycatz/tmux-rime/-/tree/main/tmux_rime)

<sup>2</sup><https://zh.wikipedia.org/w/index.php?title=%E8%81%B2%E6%AF%8D&redirect=no>

<sup>3</sup><https://en.wikipedia.org/wiki/Bopomofo>

<sup>4</sup>[https://en.wikipedia.org/wiki/Four\\_tones\\_\(Middle\\_Chinese\)](https://en.wikipedia.org/wiki/Four_tones_(Middle_Chinese))

<sup>5</sup>[https://gitlab.com/Cycatz/tmux-rime/-/blob/main/tmux\\_rime/tmux\\_rime\\_server.py](https://gitlab.com/Cycatz/tmux-rime/-/blob/main/tmux_rime/tmux_rime_server.py)

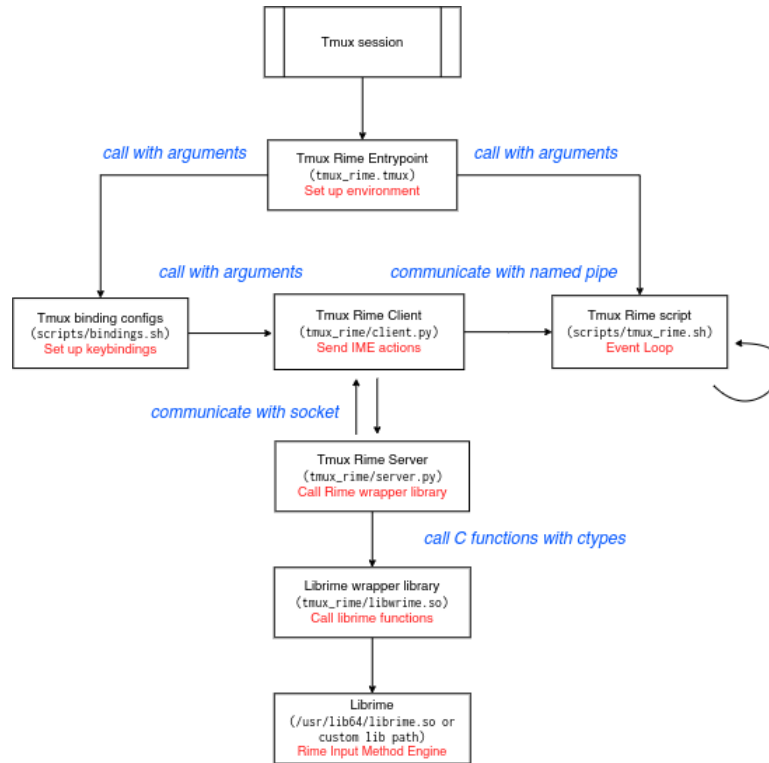


Figure 1: tmux-rime structure

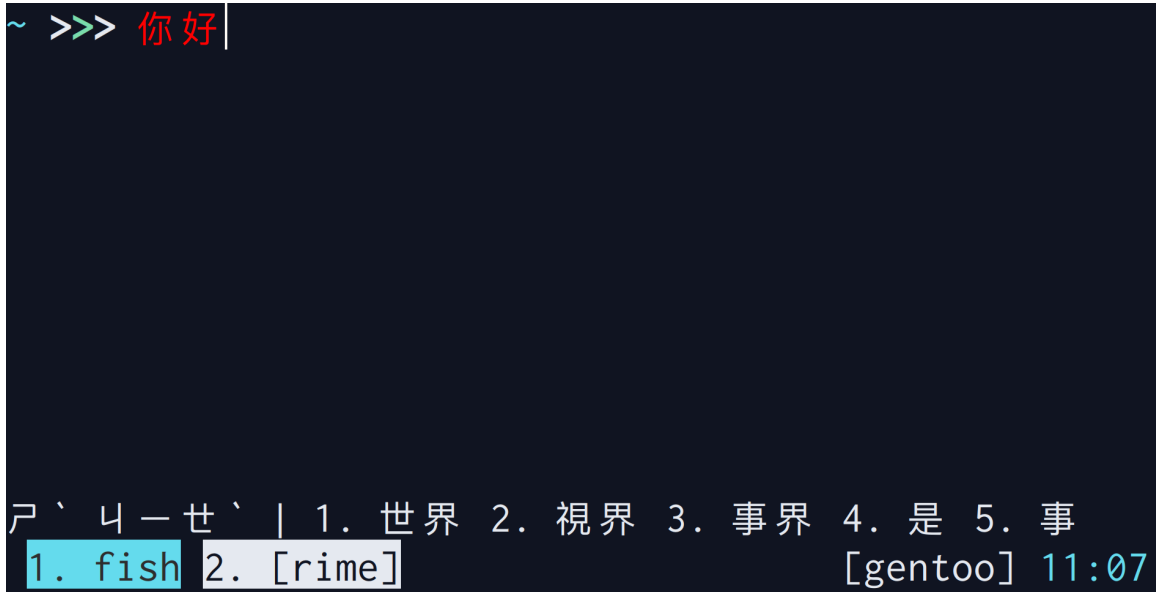


Figure 2: tmux-rime screenshot

## 2.3 Frontend

The two most important functions associated with the frontend implementation are `vt_processkey` and `terminal_input_char`. The former is for processing the key, and the latter is for inserting text onto the screen. Therefore, I hijacked the `vt_processkey` function. Instead of directly outputting text onto the screen with `terminal_input_char` function, I defined a new function called `vt_ime_process_char` to handle the user input then send requests to the IME server to convert them into CJK characters when the IME mode is enabled.

```

1  @@ -991,9 +1013,16 @@ vt_processkey(keyboard_t *kbd, struct vt_device *vd, int c)
2  + #if VT_IME
3  +         if (vt_ime_is_enabled(&vt_ime_default))
4  +             vt_ime_process_char(vw->vw_terminal, main_vd, &vt_ime_default,
5  + ↪ KEYCHAR(c));
6  +         else
7  + #endif
           terminal_input_char(vw->vw_terminal, KEYCHAR(c));

```

Another thing we need to deal with is the display of the IME status bar. The general vt display process begins with the one of vt hardware backends such as vga reading the instance of `vt_buf` content, then rendering it pixel by pixel onto the screen. However, we don't want to mess up the terminal content buffer when showing the status bar. To separate the status bar from the data buffer `vb_buffer` in struct `vt_buf`, I defined an additional member called `vb_ime_buffer` in `vt_buf` for storing the data; furthermore, the marco `VTBUF_GET_FIELD`, which serves as a helper to access the data buffer with the given row and column, also needs to be changed. I extended the macro into a inline function and added a if-else statement to access `vb_ime_buffer` only when the IME mode is enabled and the hardware is accessing at the row 0 (line 0); otherwise, read the origin buffer. That will show the status bar on the top of screen when the IME mode is enabled and hide it when disabled. The following code snippets show the patches.

```

1  @@ -212,6 +218,10 @@ struct vt_buf {
2      term_rect_t          vb_dirtyrect; /* (b) Dirty rectangle. */
3      term_char_t          *vb_buffer;   /* (u) Data buffer. */
4      term_char_t          **vb_rows;    /* (u) Array of rows */
5  +
6  + #ifdef VT_IME
7  +     term_char_t          *vb_ime_buffer; /* (u) IME status bar buffer. */
8  + #endif
9  };

```

```

1  @@ -257,8 +267,20 @@ void vtbuf_extract_marked(struct vt_buf *vb, term_char_t *buf, int
   ↪  sz);
2      ((vb)->vb_history_size)
3  #define          VTBUF_GET_ROW(vb, r) \
4      ((vb)->vb_rows[((vb)->vb_roffset + (r)) % VTBUF_MAX_HEIGHT(vb)])
5  -#define          VTBUF_GET_FIELD(vb, r, c) \
6  +
7  +#ifdef VT_IME
8  +#define VTBUF_GET_FIELD(vb, r, c) \
9      ((vb)->vb_rows[((vb)->vb_roffset + (r)) % VTBUF_MAX_HEIGHT(vb)][(c)])
10 +#else
11 +inline term_char_t VTBUF_GET_FIELD(const struct vt_buf *vb, int r, int c)
12 +{
13 +    if (vt_test && r == 0) {
14 +        return vb->vb_ime_buffer[c];
15 +    } else {
16 +        return ((vb)->vb_rows[((vb)->vb_roffset + (r)) % VTBUF_MAX_HEIGHT(vb)][(c)]);
17 +    }
18 +}
19 +#endif

```

The kernel modifications are mostly under `sys/dev/vt` directory. To avoid cluttering the origin code, I created a additional directory called `ime` to store my patches. Here is a list of static and non-static functions defined in `ime/vt_ime.{c,h}`

Static functions:

- static int vt\_ime\_send\_message(struct vt\_ime \*vi, char \*message, char \*ret)
- static int vt\_ime\_send\_char(struct vt\_ime \*vi, int ch, char \*ret)
- static int vt\_ime\_delete(struct vt\_ime \*vi, char \*ret)
- static int vt\_ime\_request\_output(struct vt\_ime \*vi, char \*ret)
- static int vt\_ime\_check\_valid\_char(struct vt\_ime \*vi, int ch)
- static int vt\_ime\_convert\_utf8\_byte(int \*utf8\_left, int \*utf8\_partial, unsigned char c)
- static void vt\_ime\_input(struct terminal \*, const void \*, size\_t)

Non-static functions:

- int vt\_ime\_toggle\_mode(struct vt\_ime \*vi)
- int vt\_ime\_is\_enabled(struct vt\_ime \*vi)
- int vt\_ime\_process\_char(struct terminal \*terminal, struct vt\_device \*vd, struct vt\_ime \*vi, int ch)
- void vt\_ime\_draw\_status\_bar(struct vt\_device \*vd, char \*status)

### 2.3.1 vt\_ime\_send\_message

Defined in: `sys/dev/vt/ime/vt_ime.c`



- `static int vt_ime_send_message(struct vt_ime *vi, char *message, char *ret)`

**Description:** for communicating with the IME server with socket.

### 2.3.2 `vt_ime_send_char`

**Defined in:** `sys/dev/vt/ime/vt_ime.c`

- `static int vt_ime_send_char(struct vt_ime *vi, int ch, char *ret)`

**Description:** for sending a single char data with `vt_ime_send_message`.

### 2.3.3 `vt_ime_delete`

**Defined in:** `sys/dev/vt/ime/vt_ime.c`

- `static int vt_ime_delete(struct vt_ime *vi, char *ret)`

**Description:** for sending the string "delete" with `vt_ime_send_message` for performing the delete action.

### 2.3.4 `vt_ime_request_output`

**Defined in:** `sys/dev/vt/ime/vt_ime.c`

- `static int vt_ime_request_output(struct vt_ime *vi, char *ret)`

**Description:** for sending the string "output" with `vt_ime_send_message` for requesting the text that will be inserted.

### 2.3.5 `vt_ime_check_valid_char`

**Defined in:** `sys/dev/vt/ime/vt_ime.c`

- `static int vt_ime_check_valid_char(struct vt_ime *vi, int ch)`

**Description:** for deciding which keys are required to be captured in the IME mode.

### 2.3.6 `vt_ime_convert_utf8_byte`

**Defined in:** `sys/dev/vt/ime/vt_ime.c`

- `static int vt_ime_convert_utf8_byte(int *utf8_left, int *utf8_partial, unsigned char c)`

**Description:** for converting a single utf8-encoded char sequence into a 32-bit unsigned integer (`term_char_t`).

### 2.3.7 `vt_ime_input`

**Defined in:** `sys/dev/vt/ime/vt_ime.c`

- `static void vt_ime_input(struct terminal *term, const void *buf, size_t len)`

**Description:** for inserting a utf8-encoded string `buf` with len `len` into the terminal with `terminal_input_char`.

### 2.3.8 **vt\_ime\_toggle\_mode**

Defined in: sys/dev/vt/ime/vt\_ime.c

- int vt\_ime\_toggle\_mode(struct vt\_ime \*vi)

Description: for toggling the IME mode.

### 2.3.9 **vt\_ime\_is\_enabled**

Defined in: sys/dev/vt/ime/vt\_ime.c

- int vt\_ime\_is\_enabled(struct vt\_ime \*vi)

Description: for checking if the IME mode is enabled.

### 2.3.10 **vt\_ime\_process\_char**

Defined in: sys/dev/vt/ime/vt\_ime.c

- int vt\_ime\_process\_char(struct terminal \*terminal, struct vt\_device \*vd, struct vt\_ime \*vi, int ch)

Description: for processing chars and performing different actions.

### 2.3.11 **vt\_ime\_draw\_status\_bar**

Defined in: sys/dev/vt/ime/vt\_ime.c

- void vt\_ime\_draw\_status\_bar(struct vt\_device \*vd, char \*status)

Description: for drawing the IME status on the screen

## 3 Conclusion