# Final Report

Fan Chung
August 19, 2021

## Contents

**1 Introduction to `vt(4)`** **1**
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
**2 My works** **4**
2.1 Introduction . . . . . 5
2.2 Backend . . . . . 5
2.3 Frontend . . . . . 7
2.3.1 `vt_ime_send_message` . . . . . 7
2.3.2 `vt_ime_send_char` . . . . . 7
2.3.3 `vt_ime_delete` . . . . . 7
2.3.4 `vt_ime_request_output` . . . . . 8
2.3.5 `vt_ime_check_valid_char` . . . . . 8
2.3.6 `vt_ime_convert_utf8_byte` . . . . . 8
2.3.7 `vt_ime_input` . . . . . 8
2.3.8 `vt_ime_toggle_mode` . . . . . 8
2.3.9 `vt_ime_is_enabled` . . . . . 8
2.3.10 `vt_ime_process_char` . . . . . 8
2.3.11 `vt_ime_draw_status_bar` . . . . . 9
**3 Conclusion** **9**

## 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.<colornum>.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 scoll 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 termianl 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 initalization.

```
/*  sys/dev/vt/vt_sysmouse.c:477 */
static void
sysmouse_drvinit(void *unused)
{

    if (!vty_enabled(VTY_VT))
        return;
    mtx_init(&sysmouse_lock, "sysmouse", NULL, MTX_DEF);
    cv_init(&sysmouse_sleep, "sysmrd");
    make_dev(&sysmouse_cdevsw, 0, UID_ROOT, GID_WHEEL, 0600,
        "sysmouse");
#ifdef EVDEV_SUPPORT
    sysmouse_evdev_init();
#endif
}

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 infomation.

```
/* sys/dev/vt/vt_sysmouse.c:202 */
void
sysmouse_process_event(mouse_info_t *mi)
{
    /* ... */

    #ifndef SC_NO_CUTPASTE
    mtx_unlock(&sysmouse_lock);
    vt_mouse_event(mi->operation, x, y, mi->u.event.id, mi->u.event.value,
        sysmouse_level);
    return;
    #endif
```

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

```c
/* src/sys/dev/vt/vt_consolectl.c:50 */
static int
consolectl_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int flag,
    struct thread *td)
{
    /* ... */

    case CONS_MOUSECTL: {
        mouse_info_t *mi = (mouse_info_t*)data;

        sysmouse_process_event(mi);
        return (0);
    }
```

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 sereval hardware backends

### 1.3.3 Terminal Features

```c
/* sys/dev/vt/vt_core.c:90 */
const struct terminal_class vt_termclass = {
  .tc_bell = vtterm_bell,
  .tc_cursor = vtterm_cursor,
  .tc_putchar = vtterm_putchar,
  .tc_fill = vtterm_fill,
  .tc_copy = vtterm_copy,
  .tc_pre_input = vtterm_pre_input,
  .tc_post_input = vtterm_post_input,
  .tc_param = vtterm_param,
  .tc_done = vtterm_done,

  .tc_cnprobe = vtterm_cnprobe,
  .tc_cngetc = vtterm_cngetc,

  .tc_cngrab = vtterm_cngrab,
  .tc_cnungrab = vtterm_cnungrab,

  .tc_opened = vtterm_opened,
  .tc_ioctl = vtterm_ioctl,
  .tc_mmap = vtterm_mmap,
};
```

# 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, recieves 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 developmemt 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`[1] 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 recieve 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*[2] (聲母 in chinese) 「ㄓ」(chih in *Wade–Giles* romanization) in the input schema *Bopomofo*[3] , 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,[4] such as pressing the key '3' to compose 「ㄓˇ」or the space key to compose 「ㄓ」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`[5]

Before implementing the frontend in `vt`, I implemented the frontend for `tmux` 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`.

[1] https://gitlab.com/Cycatz/tmux-rime/-/tree/main/tmux_rime
[2] https://zh.wikipedia.org/w/index.php?title=%E8%81%B2%E6%AF%8D&redirect=no
[3] https://en.wikipedia.org/wiki/Bopomofo
[4] https://en.wikipedia.org/wiki/Four_tones_(Middle_Chinese)
[5] https://gitlab.com/Cycatz/tmux-rime/-/blob/main/tmux_rime/tmux_rime_server.py

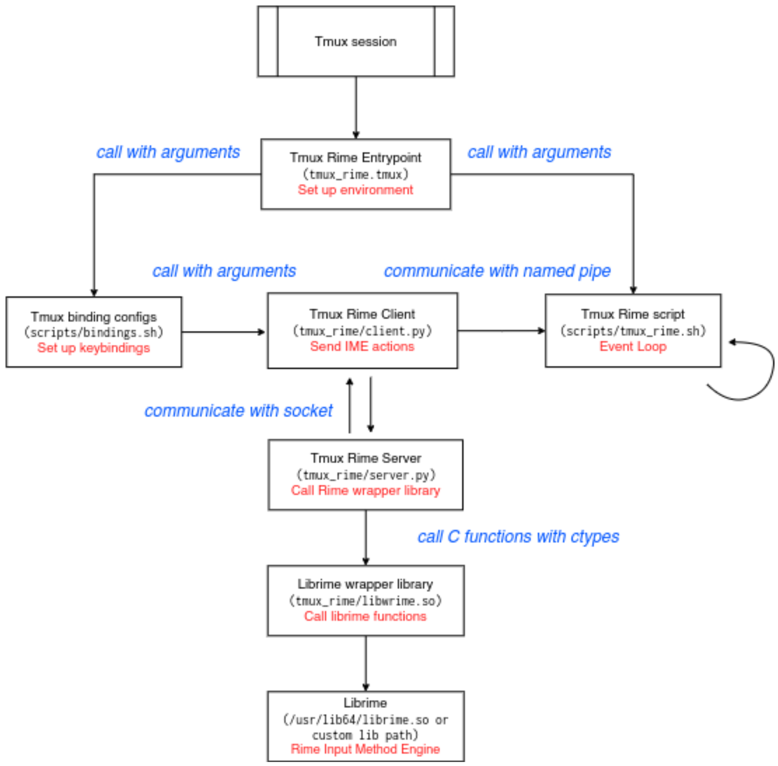


Figure 1: tmux-rime structure

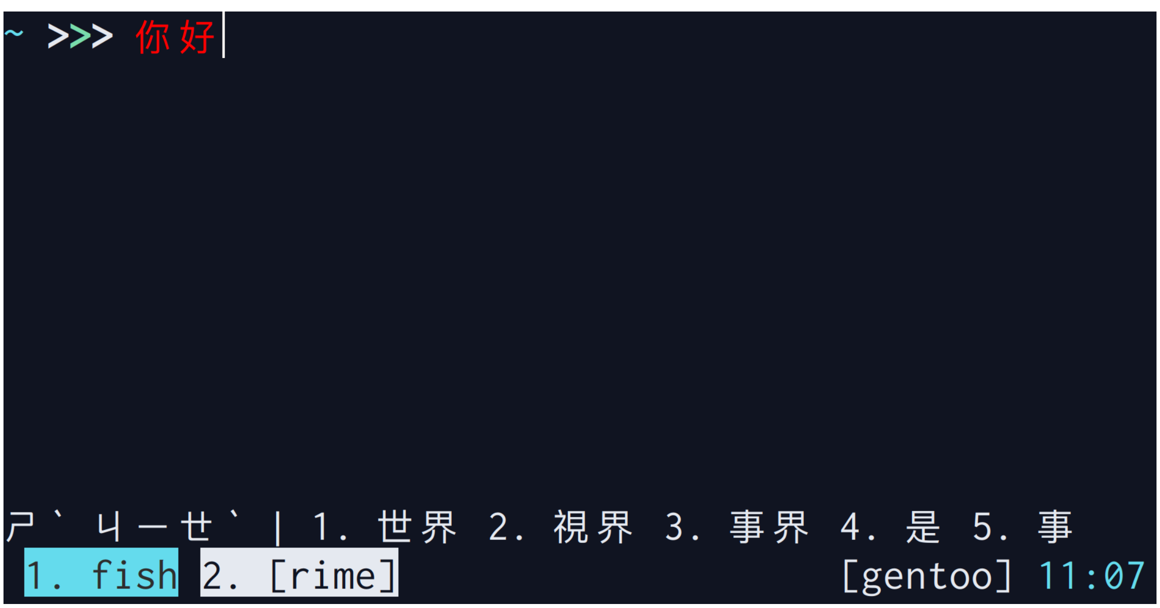


Figure 2: tmux-rime screenshot

## 2.3 Frontend

The 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