Input method in FreeBSD virtual terminal

Fan Chung

August 19, 2021

Contents

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 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 「ㄓ ˇ」 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` 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`.

2.1 `tmux-rime` screenshots

[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

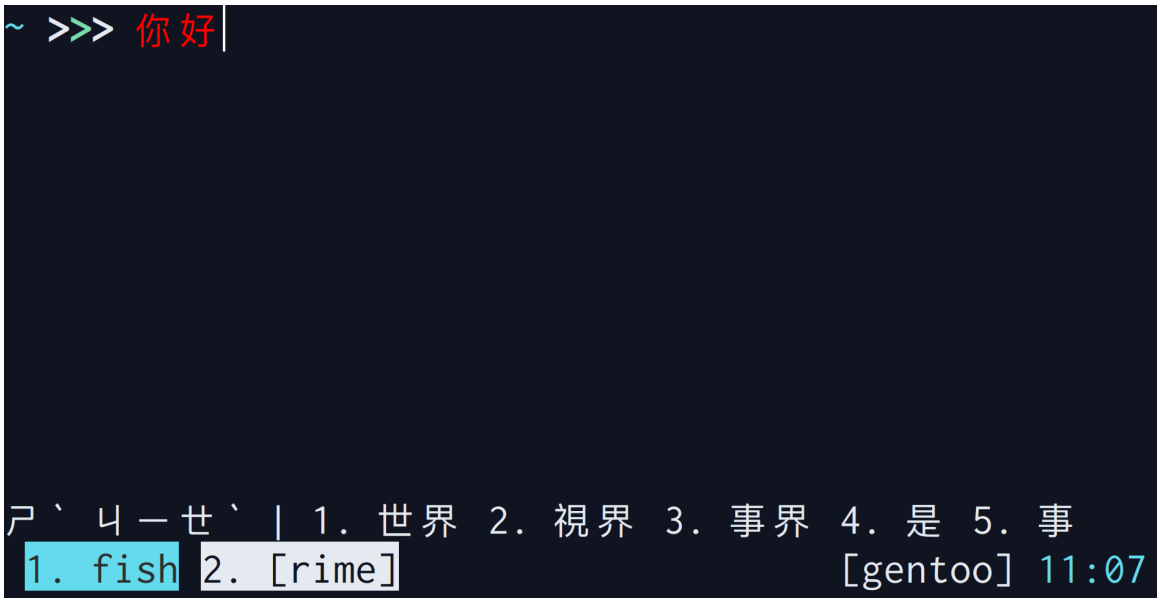

Figure 2: tmux-rime screenshot

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.

```
@@ -991,9 +1013,16 @@ vt_processkey(keyboard_t *kbd, struct vt_device *vd, int c)
+ #if VT_IME
+             if (vt_ime_is_enabled(&vt_ime_default))
+                 vt_ime_process_char(vw->vw_terminal, main_vd, &vt_ime_default,
↪  KEYCHAR(c));
+             else
+ #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; futhermore, 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.

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

```
@@ -257,8 +267,20 @@ void vtbuf_extract_marked(struct vt_buf *vb, term_char_t *buf, int
↪  sz);
        ((vb)->vb_history_size)
 #define        VTBUF_GET_ROW(vb, r) \
        ((vb)->vb_rows[((vb)->vb_roffset + (r)) % VTBUF_MAX_HEIGHT(vb)])
-#define        VTBUF_GET_FIELD(vb, r, c) \
+
+#ifdef VT_IME
+#define VTBUF_GET_FIELD(vb, r, c) \
        ((vb)->vb_rows[((vb)->vb_roffset + (r)) % VTBUF_MAX_HEIGHT(vb)][(c)])
+#else
+inline term_char_t VTBUF_GET_FIELD(const struct vt_buf *vb, int r, int c)
+{
+    if (vt_test && r == 0) {
+        return vb->vb_ime_buffer[c];
+    } else {
+        return ((vb)->vb_rows[((vb)->vb_roffset + (r)) % VTBUF_MAX_HEIGHT(vb)][(c)]);
+    }
+}
+#endif
```

In addition to the kernel modifications mentioned above, I also defined several custom functions to implement the frontend. To avoid cluttering the origin code, I created a additional directory called `ime` under `sys/dev/vt` to store my patches. You may refer to Appendix for detailed documentation.

4 Screenshots

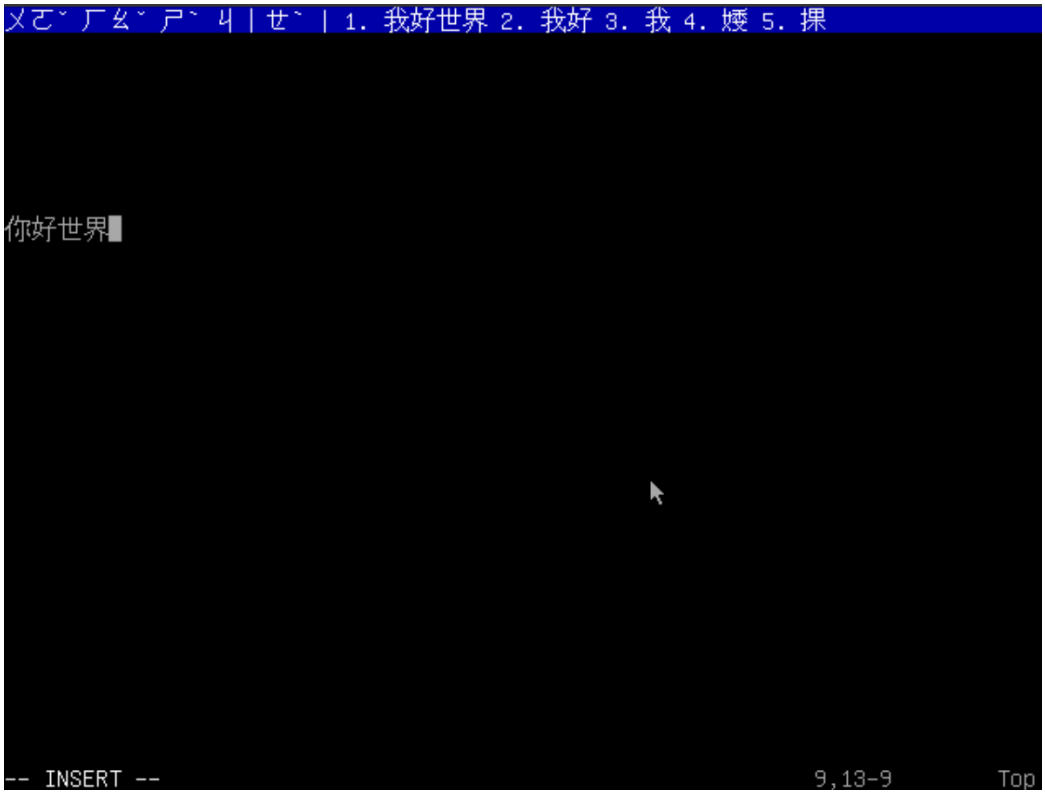

Figure 3: Screenshot of running IME in vt

5 Conclusion

5.1 Thought

5.2 Future work

6 Appendix

6.1 Introduction

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)`

6.2 `vt_ime_send_message`

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

Description: for communicating with the IME server with socket.

6.3 `vt_ime_send_char`

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

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

6.4 `vt_ime_delete`

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

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

6.5 vt_ime_request_output

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

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

6.6 vt_ime_check_valid_char

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

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

6.7 vt_ime_convert_utf8_byte

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

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

6.8 vt_ime_input

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

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

6.9 vt_ime_toggle_mode

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

Description: for toggling the IME mode.

6.10 vt_ime_is_enabled

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

Description: for checking if the IME mode is enabled.

6.11 vt_ime_process_char

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

Description: for processing chars and performing different actions.

6.12 vt_ime_draw_status_bar

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

Description: for drawing the IME status on the screen