things I learned about computers while writing a text editor: screen display

. . . it all started with a crash!

the answer was, obviously, to write my own text editor!

first talk

the control key !

second talk

the raw mode vs canonical mode !

this talk. . .

how to tell the computer to display something on the screen?

this talk. . .

well, just refresh the display after every keypress!

wait, what?

JUST REFRESH THE DISPLAY EVERY KEYPRESS???

breaking mental models

the long road to “yes, I know how a computer works!”

a new mental model

so although we can control every pixel, apparently this is not the way!

some code

```
if __name__ == "__main__":

    [...]

    while True:
        editor.refresh_screen()
        editor.process_keypress()
        editor.autosave()
```

more code

the key feature in refresh_screen():

`os.write(fd, buffer_string)`

docs.python.org: Write the bytestring in str to file descriptor fd.

more code!!!

how is os.write implemented ??

to be continued. . .

to zulip !

. . . but how is the screen refreshed. . . ‘generally speaking’?

to os programming on RPi!

I guess I will need to write an OS to better understand this!!

I need a tutorial - here is one with Raspberry Pi:

$=>$ an open source tutorial!!

to using the screen on the RPi OS !

there is a mailbox!

you communicate to the GPU by leaving it small messages, appropriately encoded!

and you loop on an image frame buffer. . . ?!

code I am working on understanding, basically you are reading images pixel by pixel and displaying them.

THANK YOU!!!!

to asm

"The first thing we are going to need to program is a 'postman'. This is just two methods: MailboxRead, reading one message from the mailbox channel in r0. and MailboxWrite, writing the value in the top 28 bits of r0 to the mailbox channel in r1. The Raspberry Pi has 7 mailbox channels for communication with the graphics processor, only the first of which is useful to us, as it is for negotiating the frame buffer."

from => the previously mentioned tutorial

to asm

```
.globl GetMailboxBase
GetMailboxBase:
    ldr r0,=0x2000B880
    mov pc,lr
```

to asm !!

`ldr reg,=val` moves the value `val` to the register `reg`.

to asm !!

"`mov pc,lr` copies the value in lr to pc. As mentioned earlier lr always contains the address of the code that we have to go back to when a method finishes. pc is a special register which always contains the address of the next instruction to be run. A normal branch command just changes the value of this register. By copying the value in lr to pc we just change the next line to be run to be the one we were told to go back to."

to asm

```
.globl GetMailboxBase
GetMailboxBase:
    ldr r0,=0x2000B880
    mov pc,lr
```

more code

```
.globl MailboxWrite
MailboxWrite:
[...]
```

more code

```
.globl MailboxRead
MailboxRead:
[...]
```