The Oldest Linux Peripheral

Matt Mullins

v1

The Museum

7000 E Marginal Way S
Seattle, WA 98108

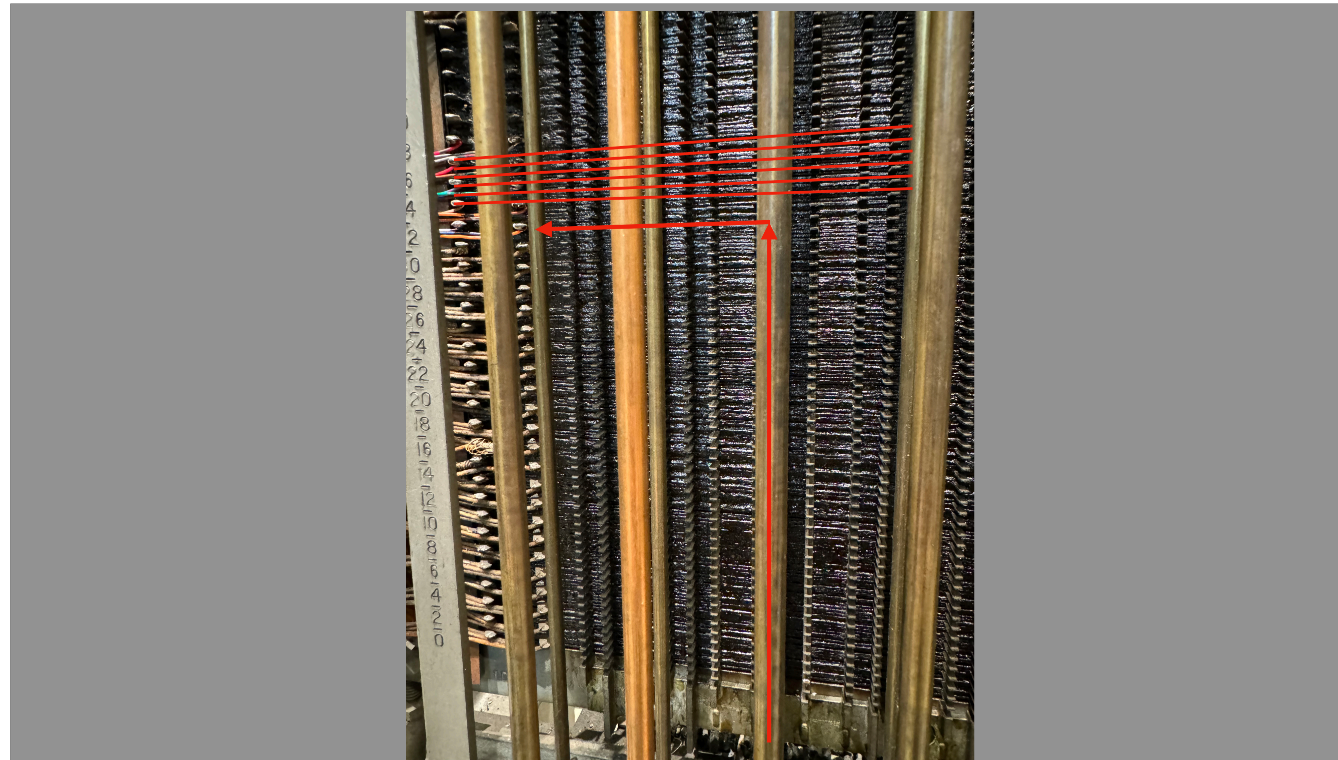
The Connections Museum in Seattle has a variety of **functioning** telephone switching equipment — the stuff usually hidden away in a nondescript brick building downtown.

Our collection starts with the Step-by-Step switches, based on technology developed by an undertaker in Kansas City with patents dating back to 1891. We then have a series of Western Electric electromechanical switching equipment, taking you through the Panel, #1 Crossbar, and #5 Crossbar.

On our second floor, we have our computer-controlled (#3 ESS) and fully digital switches (Nortel DMS-10).

The Panel

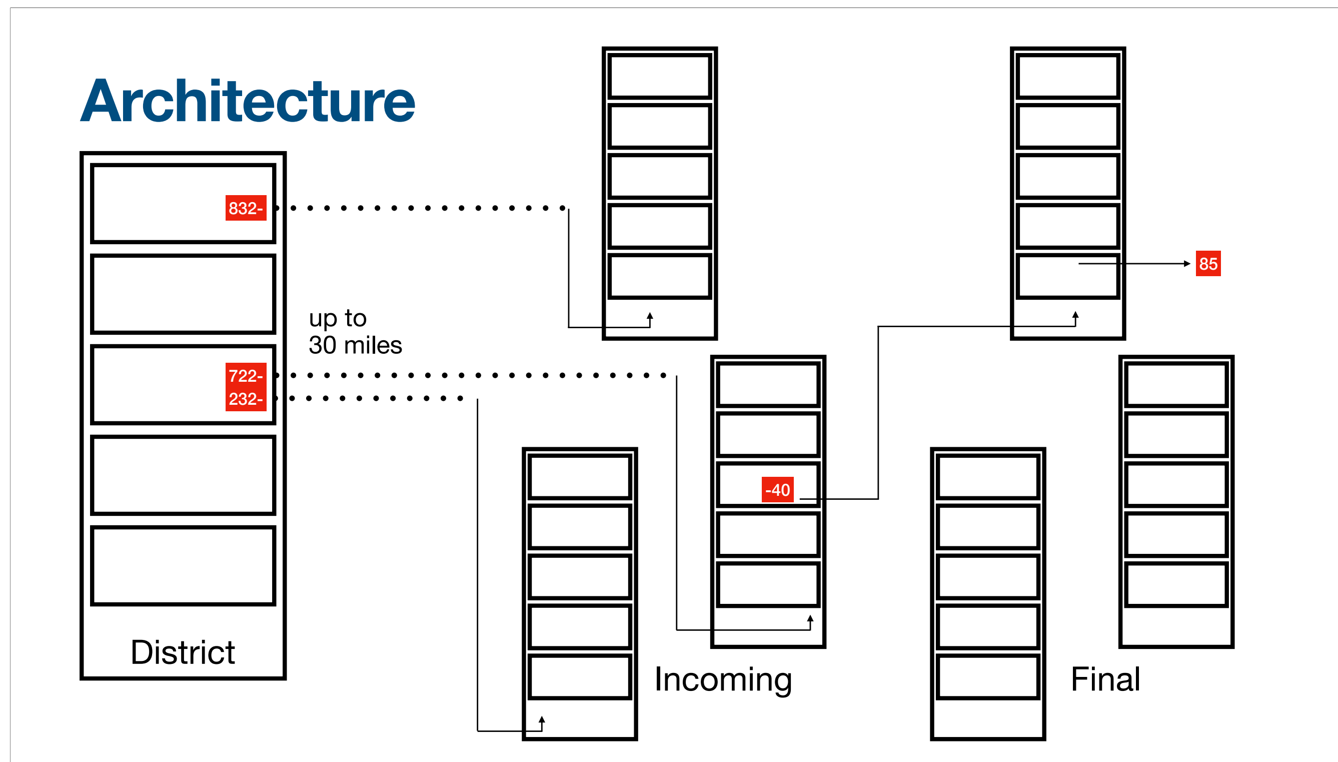
The Panel switch is the focus of this presentation. And this is Sarah, who takes care of oiling, dusting, adjusting, repairing, and expanding this whole machine.



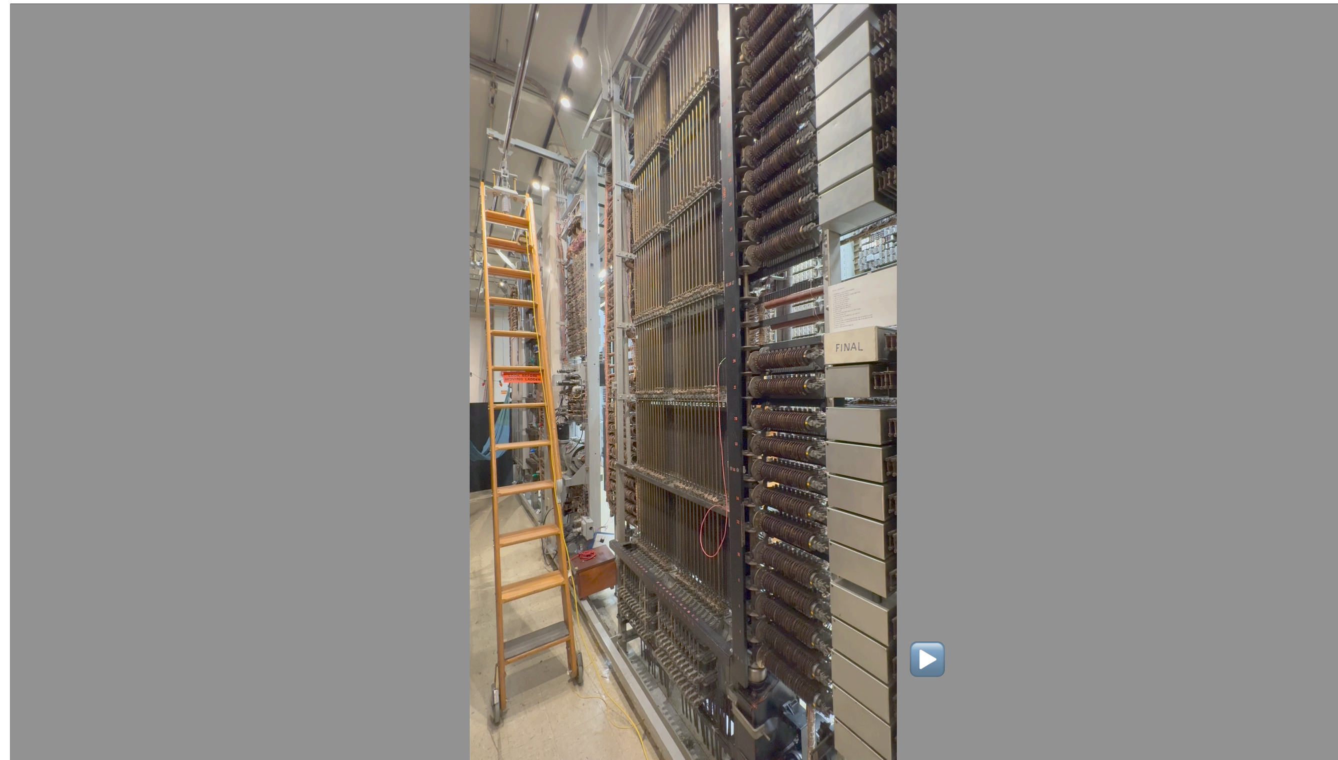
One "bank" of the panel consists of 100 circuits' worth of metal contacts that span the whole width of the frame, allowing any one of the vertical rods to direct a call to the appropriate next-hop.

You can perhaps see that the wires toward the top are more colorful — that's a sure sign that this is new, functioning wiring rather than the older cloth-covered wiring that is original to the machine.

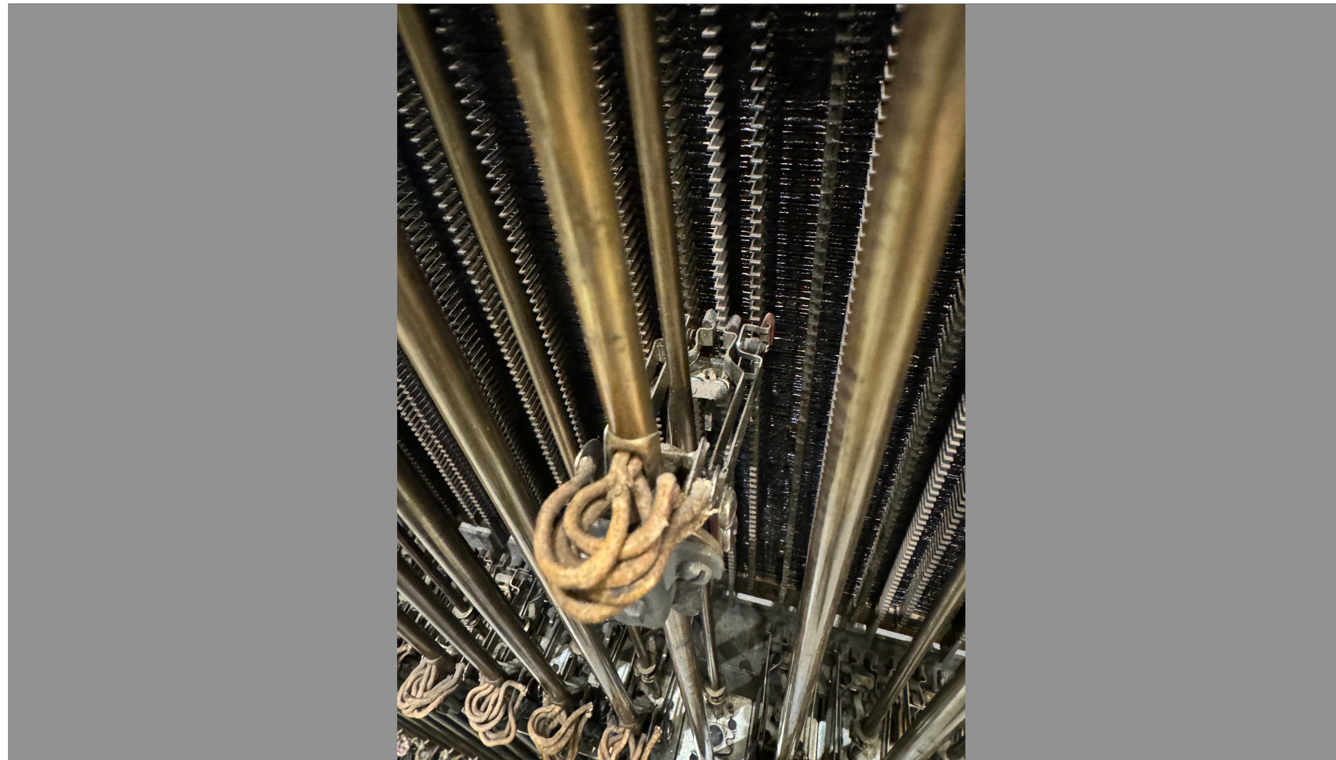
Here's a a whole bank of contacts for scale; there are five banks in every frame (i.e. rack).



A call progresses through **several** of these frames on its way to the end-user. Each layer lets 30 inputs take one of 500 paths out, forming a tree hierarchy of possible routes.

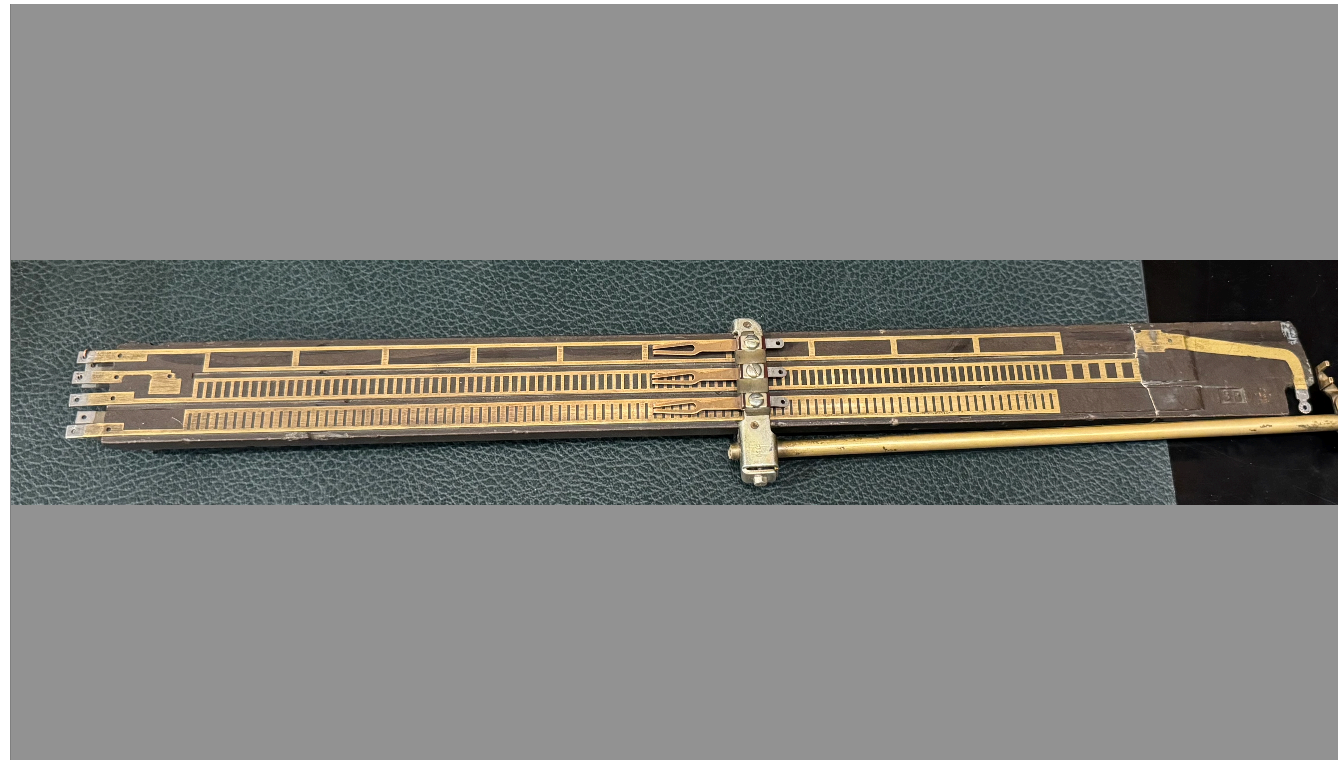


Here's a video of a rod moving up to make the last step in selecting an individual customer's line to ring.



and up close, so you can see the brush that moves up to make contact with the desired output circuit.

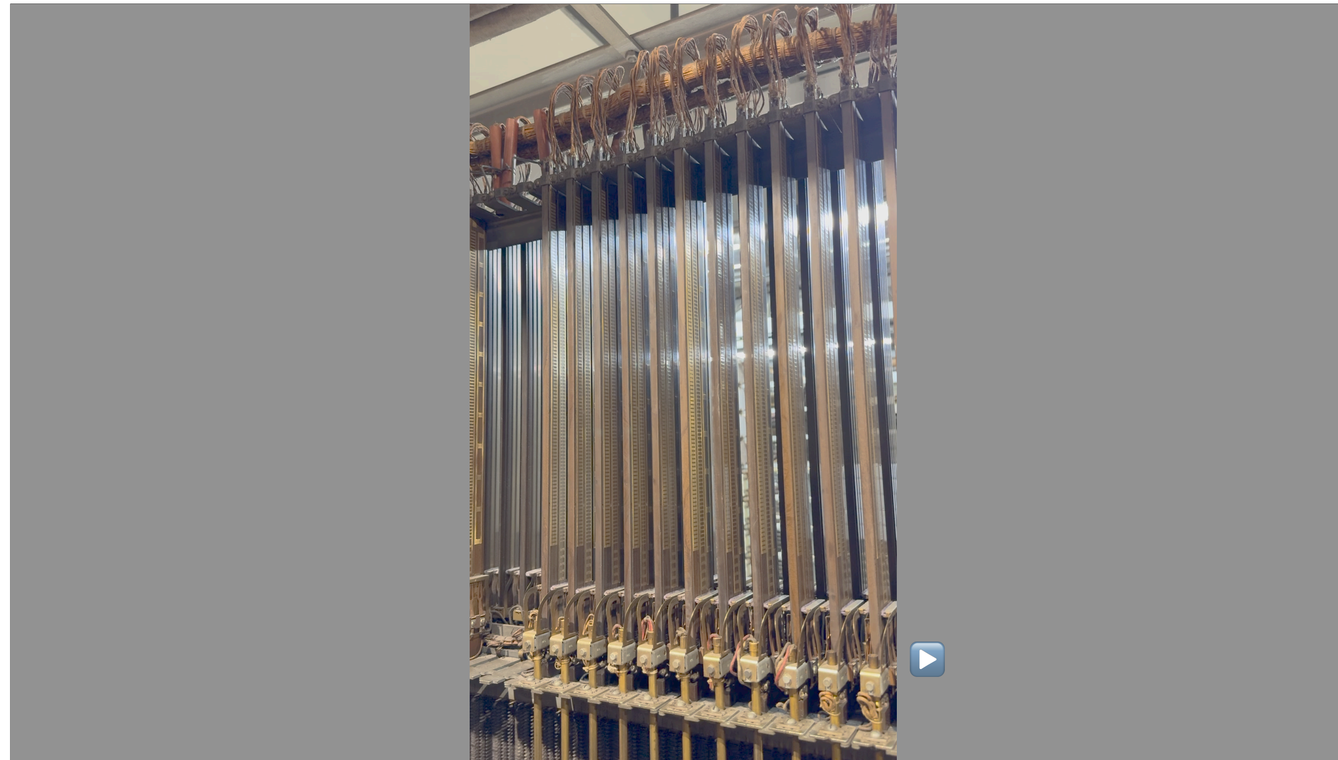
Those cloth wires run through the inside of the rod, to take the call from the input to this brush.



The top of the rod is connected to something a little different, though — this commutator is how the "sender" (the control circuitry) can know exactly how far the rod has moved.

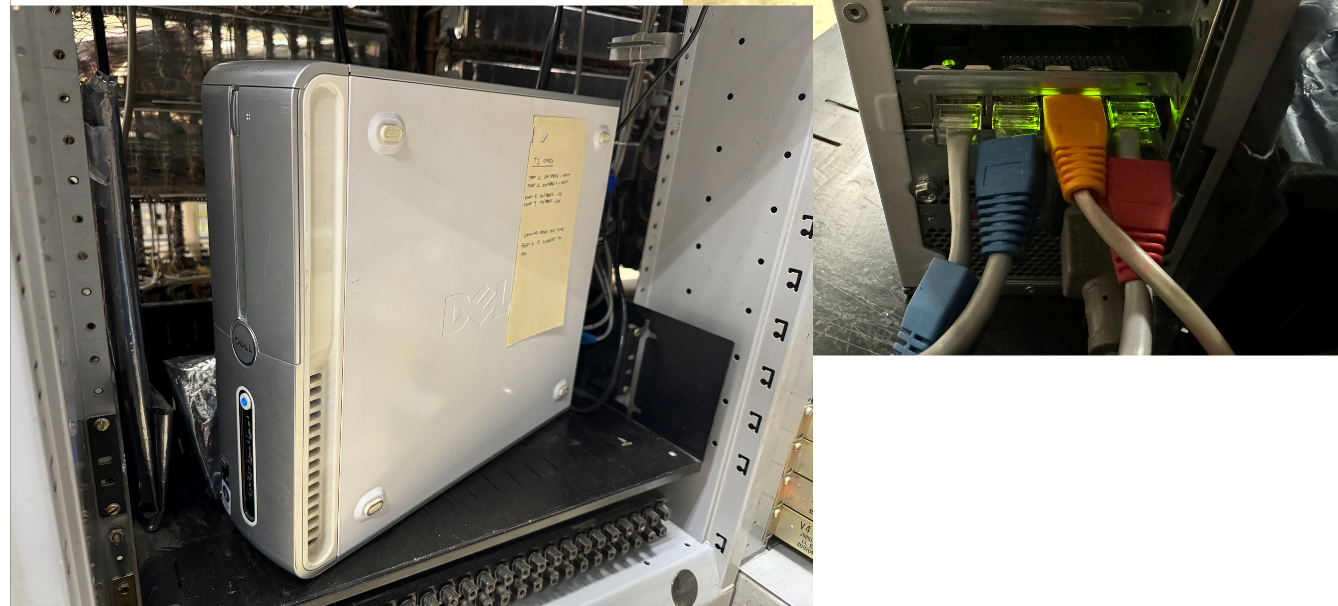
Notice the ten widely-spaced contacts across the top brush (which would be toward the rear of the frame when this is mounted vertically); that moves to the tens-digit the caller dialed. Then the machine counts pulses from the middle row of contacts, which let it choose the exact recipient of the call.

These electrical pulses move **backwards**, from the moving equipment to the controlling equipment, hence its name "revertive pulse". The sender doesn't tell the rod where to move: it can only say "start moving", wait to see what happens, then "stop!".



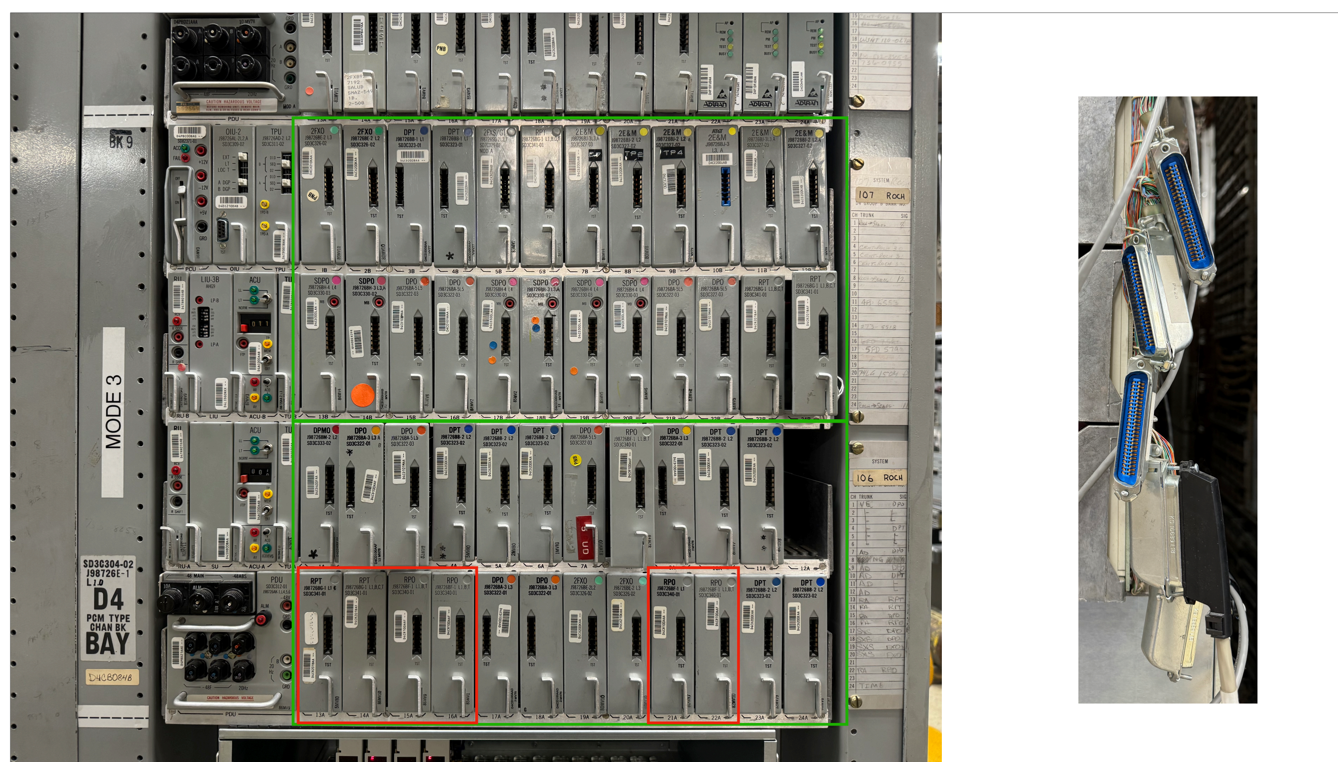
Here you can see this brush moving across the commutator while a call is being placed.

The computer



OK, but how is this computer related? One of our goals is to keep the equipment moving — this knocks the dust out of contacts, and makes sometimes horrific noises to let us know when it's time to make an adjustment.

This whole thing runs on the world's worst Dell workstation, simply outfitted with a PCIe quad-T1 card (Sangoma A104).

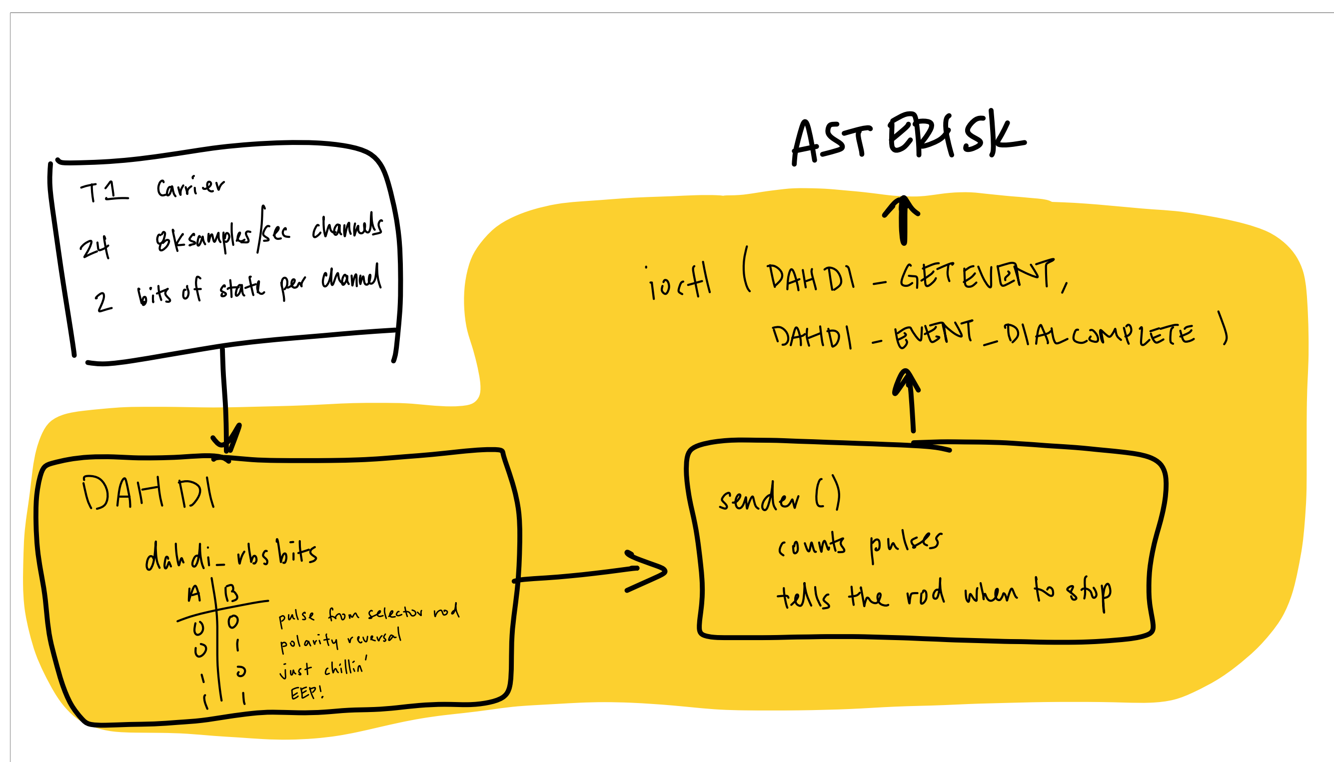


The other end of that T1 circuit is this, our D4 channel bank. One unit turns two T1 circuits into a ... complete pile of copper connections in the picture on the right; each channel has anywhere between two and six pins that convey its state to the analog machinery.

Each of the green boxes are the cards to service a single T1 circuit, which contains 24 digital channels.

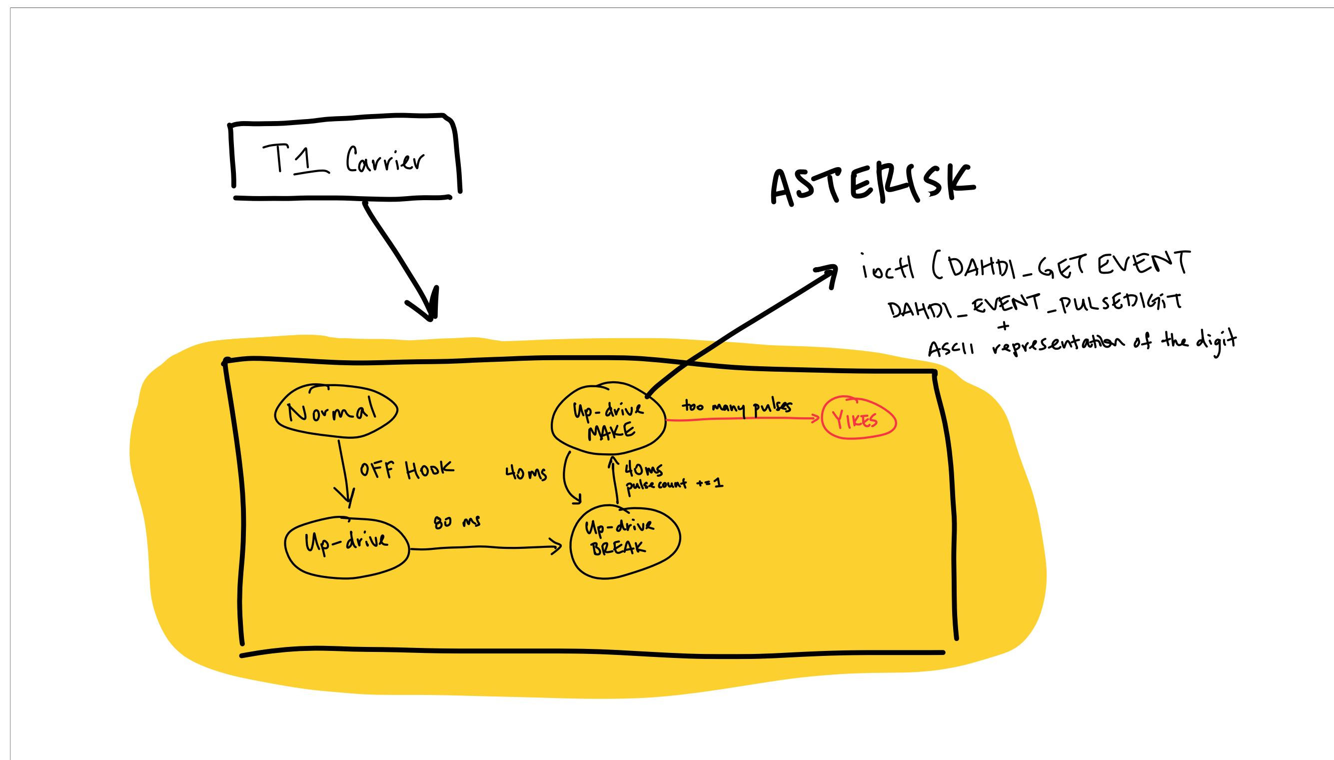
The channels we're interested in are highlighted in red — "RPT" and "RPO" for Revertive Pulse Terminating (or Originating).

Linux



So why does this need kernel support? DAHDI — the Digium Asterisk Hardware Device Interface — does the bulk of its logic in kernel-space. Nowadays this would be considered an antipattern, but when this was designed it was reasonable to do this since there is a real-time requirement.

For placing a call, Asterisk tells DAHDI to place a call to a 4-digit number; the kernel converts this to exactly where it wants the "incoming" and "final" rods to go, then counts the pulses it gets back from each selector rod and eventually tells Asterisk simply that "dialing is complete!"



Counting pulses that are coming from the rods is comparatively easy; in order to have the Panel make calls through Asterisk, Asterisk instead has to **pretend to be** one of those selector rods. DAHDI already has a multi-layer convoluted state machine to handle tasks like this, so Sarah added the additional states and state transitions we needed to control Revertive Pulse.

This code is chock full of pretending to be an electromechanical machine with spinny motors, gears, brushes, relays, etc. That 80ms delay before we even get started? That's to emulate the physical delay it would have taken to get a selector moving in the first place — the sender **depends** on this or it's going to miss pulses! She took care to comment the code with tons of references to the Circuit Description documents from when the machine was created to explain all of this behavior.

Kernel Debugging

- bpftrace is AWESOME

```
sudo snap run bpftrace -p $(pidof asterisk) -e '
    kprobe:sys_ioctl { printf("0x%x\n%s\n", arg1, ustack); }'

sudo snap run bpftrace -e '
    kprobe:dahdi_chan_ioctl /arg1 == 0x4004da07/ {
       printf("ptr: %p, ioctl: 0x%x, arg: %d\n%s\n",
       arg0, arg1, *(int32*)uptr(arg2), ustack); }'
```

- a **whole lot** of module_printk().

The process of building this was arduous. It is tough to look at something as large as Asterisk **and then also** the hardware interface layer and figure out what codepaths will be hit; this is compounded by Asterisk having two copies of some routines (e.g. `getsigstr()` is in both `sig_analog.c` and `chan_dahdi.c`). bpftrace was **indispensable** trying to figure out what-was-called-and-when; these examples here figure out what ioctls Asterisk is actually calling.

The second one in particular is looking for when Asterisk sets the hookswitch state, and can reach through and read userspace memory to find what actual state Asterisk was trying to set, and also the whole stack trace within Asterisk's code that led to the `ioctl()`.

And of course there were a lot of kernel panics along the way, when we reload the kernel module containing all this logic and find out we've made a wee bug somewhere.

An Actual Call

Kernel logs - calling ~~722~~-4085

```
0.000_000  dahdi: ioctl_dahdi_dial:  REPLACED: 0020085
0.189_745  dahdi: Pulse: 0, SELECTION: 2,  i-16960
0.222_742  dahdi: Pulse: 1, SELECTION: 2,  i-16432
0.255_720  dahdi: Pulse: 2, SELECTION: 2,  i-15904
0.255_722  dahdi:       STOP

0.607_763  dahdi: Pulse: 0, SELECTION: 3,  i-14518
0.607_765  dahdi:       STOP

1.157_739  dahdi: Pulse: 0, SELECTION: 4,  i-11350
1.157_741  dahdi:       STOP

1.586_694  dahdi: Pulse: 0, SELECTION: 5,  i-13286
1.751_719  dahdi: Pulse: 1, SELECTION: 5,  i-10646
1.916_743  dahdi: Pulse: 2, SELECTION: 5,  i-8006
2.081_752  dahdi: Pulse: 3, SELECTION: 5,  i-5366
2.257_733  dahdi: Pulse: 4, SELECTION: 5,  i-2550
```

```
2.422_711  dahdi: Pulse: 5, SELECTION: 5,  i-0
2.587_782  dahdi: Pulse: 6, SELECTION: 5,  i-0
2.763_748  dahdi: Pulse: 7, SELECTION: 5,  i-0
2.928_735  dahdi: Pulse: 8, SELECTION: 5,  i-0
2.928_737  dahdi:       STOP

3.159_693  dahdi: Pulse: 0, SELECTION: 6,  i-16454
3.247_756  dahdi: Pulse: 1, SELECTION: 6,  i-15046
3.313_764  dahdi: Pulse: 2, SELECTION: 6,  i-13990
3.379_755  dahdi: Pulse: 3, SELECTION: 6,  i-12934
3.445_777  dahdi: Pulse: 4, SELECTION: 6,  i-11878
3.511_749  dahdi: Pulse: 5, SELECTION: 6,  i-10822
3.511_751  dahdi:       STOP

3.753_784  dahdi: REVERSAL
3.753_787  dahdi: SELECTIONS COMPLETE!
```

Here we see kernel logs from Asterisk making a call; the "722" and "00" are not actually sent over the wire because it's assumed that if a call is routed on **this** wire, then it's bound for 722-. A call going anywhere else would pick a **different** set of wires.

We need to route the call through the third Incoming Brush (groups of 2,000 lines), the first Incoming Group (500 lines) on that brush. And then the first Final Brush (100 lines), the eighties for Final Tens (10 lines), and last, five for Final Units (one exact customer).

You can see the printk() from every time we receive a pulse — notice that "SELECTION: 5" is about 170ms between pulses, but "SELECTION: 6" is only about 66ms between pulses. That is because the rod has to physically move further to make the selection!

Another Call

from the Panel to 722-5234

```
0.000_000  dahdi: RPT got OFFHOOK, selidx: 0
0.176_003  dahdi: GOT ONHOOK, selidx: 0, seqswitch 1
0.176_007  dahdi: selidx 0 stored pulsecount 1

0.517_027  dahdi: RPT got OFFHOOK, selidx: 1
1.012_028  dahdi: GOT ONHOOK, selidx: 1, seqswitch 1
1.012_031  dahdi: selidx 1 stored pulsecount 5

1.474_019  dahdi: RPT got OFFHOOK, selidx: 2
1.727_008  dahdi: GOT ONHOOK, selidx: 2, seqswitch 1
1.727_012  dahdi: selidx 2 stored pulsecount 2

1.914_028  dahdi: RPT got OFFHOOK, selidx: 3
2.167_059  dahdi: GOT ONHOOK, selidx: 3, seqswitch 1
2.167_062  dahdi: selidx 3 stored pulsecount 2

2.332_047  dahdi: RPT got OFFHOOK, selidx: 4
2.585_078  dahdi: GOT ONHOOK, selidx: 4, seqswitch 1
2.585_082  dahdi: selidx 4 stored pulsecount 2
```

```
2.750_046  dahdi: RPT got OFFHOOK, selidx: 5
3.091_031  dahdi: GOT ONHOOK, selidx: 5, seqswitch 1
3.091_034  dahdi: selidx 5 stored pulsecount 3

3.256_145  dahdi: RPT got OFFHOOK, selidx: 6
3.674_018  dahdi: GOT ONHOOK, selidx: 6, seqswitch 1
3.674_022  dahdi: selidx 6 stored pulsecount 4

3.674_171  dahdi: Got an IA wink from Asterisk,
advance sequence switch.
4.015_064  dahdi: RPT got OFFHOOK, selidx: 7
4.015_066  dahdi: Finished FU selections. Sending
reversal!
4.015_068  dahdi: Selections: 1,5,2,2,2,3,4


[2024-03-21 16:34:32] DEBUG[1940][C-00008bb9]
sig_analog.c: Evaluating selections: 1,5,2,2,2,3,4

[2024-03-21 16:34:32] DEBUG[1940][C-00008bb9]
sig_analog.c: Line number is: 155234
```

And here we are in the other direction — since these pulses are being created by a computer, we aren't stuck with pesky physical limitations, so all of them are 80ms. Notice that selidx 0 took 176ms (just enough time for that 80ms wait, and one 80ms pulse), but selidx 1 took 495ms (the 80ms wait, then **five** 80ms pulses).

Both this call and the previous slide took about four seconds to fully convey all of the information needed to route the call.

Asterisk Call Routing

A Complete Mess

```
/* Do some RP math to convert selections to a phone number */

lineno[2] = selections[6];      // units
lineno[1] = selections[5];      // tens
lineno[0] = selections[4];      // Final Brush - hundreds

// Incoming Group - hundreds
lineno[0] = lineno[0] + selections[3] * 5;

// Incoming Brush - thousands
lineno[0] = lineno[0] + selections[2] * 20;

snprintf(exten, sizeof(exten)-1, "%d%d%02d%d%d",
                      selections[0], selections[1],
                      lineno[0], lineno[1], lineno[2]);
```

```
; Tandem Office selections to panel
exten => _15XXXX,1,Answer()
 same => n,Dial(DAHDI/g3/${EXTEN:2})
 same => n,Hangup()
```

Once DAHDI collects all those pulses, we also have custom code running in Asterisk (userspace) to turn these physical-layout selections into "which phone number were they calling?" — tens and units are easy, but the groups of 500 and 2000 lines take a bit more care.

After that, it just becomes an extension config just like any other configuration in Asterisk. In this case, we strip the first two digits and tell trunk group 3 that there's a call for "5234".

March 3, 1923

How old is it?

Thanks!

https://github.com/connectionsmuseum/

https://www.telcomhistory.org/

info@connectionsmuseum.org

mmullins@mmlx.us