Ioctl(2) is so 1980ies...

Poul-Henning Kamp

phk@FreeBSD.org

Kernel Hacker

What is ioctl(2)

- Ioctl(2) is the 6th system call for files in UNIX
  - Open(2), close(2)
  - Read(2), write(2)
  - Lseek(2)
  - Ioctl(2)
- From the manpage:
  - ”ioctl -- control device”

What is ioctl(2) used for ?

- ”control device”
  - Set bit-rate on serial ports.
  - Tell tape-station to rewind tape.
  - Format disk.
  - Pass DVD/DCESS key to drive.
  - Configure network interfaces.
  - (Re)define ATA-raid layout.
- The kitchen sink.

Nothing important of course...

- Loosing data
  - Format disk, erase tape etc.
- Destroy Hardware
  - Setting bogus parameters
- Make system unusable in various ways
  - Panic(8) implementations.
  - Set SLIP linedisc on console.

Kitchensink arguments

- ioctl(int d, unsigned long request, ...);
  - Request = magic number
  - ... = ”something”
- Type-checking is a town in Russia.
- Magic number collisions.
  - SLIOCSKEEPAL == PPPIOCSRASYNCMAP
  - Not that much of a problem.
    - until you use the wrong program on a device.

The 3BSD situation (1980)

tty.h:

```
#define TIOCSETC        (('t'<<8)|17)
#define TIOCGETC        (('t'<<8)|18)
```

userland:

```
e = ioctl(fd, TIOCSETC, &tc);
```

kernel:

```
case TIOCSETC:
        if (copyin(addr, (caddr_t)&tun, sizeof(struct tc)))
                u.u_error = EFAULT;
        break;

case TIOCGETC:
        if (copyout((caddr_t)&tun, addr, sizeof(struct tc)))
                u.u_error = EFAULT;
        break;
```

CSRG ports UNIX to 32 bits

Being smart, the 1980ies way

- Move to VAX gives request 16 extra bits.
  - Use them for generic handling:
    - 1 bit Copy args in.
    - 1 bit Copy args out.
    - 1 bit Don't copy args.
    - 13 bits Length of args.
  - Retain bottom 16 bits compatible:
    - 8 bits Group (typically ASCII char).
    - 8 bits Number (typically integer).

The 4.2BSD situation (1983)

tty.h:

```
#define TIOCSETC  _IOW(t,17,struct tchars)      /* set special characters */
#define TIOCGETC  _IOR(t,18,struct tchars)      /* get special characters */
```

userland:

```
e = ioctl(fd, TIOCSETC, tc);
```

kernel:

```
case TIOCGETC:
        bcopy((caddr_t)&tp->t_intrc, data, sizeof (struct tchars));
        break;

case TIOCSETC:
        bcopy(data, (caddr_t)&tp->t_intrc, sizeof (struct tchars));
        break;
```

Banned or doomed.

- How do we design the API ?
  - Struct foo_control reflect the hardware bits.
  - Struct foo_control is abstract representation.

Hardware representation

- Good sides:
  - Easy to prototype and fiddle hardware.
  - Small amount of code in kernel.
- Bad sides:
  - People tend to skip parameter validation.
  - Mk II controller will have different registers.
  - Puts hardware knowledge in userland.
    - UNIX is all about not doing that.

Abstract representation

- Good sides:
  - Encourages sanity checks
  - Provides hardware independent API/ABI
- Bad sides:
  - Takes de-abstraction code in the kernel.
  - Generalizing from 1 instance.
  - Still does not cope well with Mk II hardware.

Diminishing return...

- Ioctls are mainly used administratively.
- Administrative operations happen seldom.
- Flexible hardware -> many ioctls.
- Many ioctls -> much code.
- Much code seldom used -> less testing.
- QED: more bugs & security issues.

Public API/ABI location ?

- Where is the public API/ABI for the device ?
- Is it the ioctls ?
  - Requires argument checking, security.
- Is it the foocontrol(8) program ?
  - Does that mean we do not need to check ?

The True UNIX spirit: DDTT (?)

- The argument goes something like:
  - We make sure only root can do this ioctl.
    - No security issues.
  - We provide a program to do so: foocontrol.
  - No other program should use the ioctl.
  - If people complain, we tell them:
    - Don't Do That Then!
- Ioctl calls are not a supported API/ABI.

Pseudo code...

include/fooio.h:

```
struct foo_control {
};
#define FOOBAR   _IOC('F', 23, struct foo_control)
```

sbin/foocontrol/foocontrol.c:

```
Main()
{
        Process arguments
        check that they make sense
        parse, interpret and pack into struct foo_control
        error = ioctl(fd, FOOBAR, fc);
}
```

sys/dev/foo.c:

```
foo_ioctl(...)
{
case FOOBAR:
        /* XXX: should check permissions */
        /* XXX: should check arguments */
        Unpack struct foo_control and apply
}
```

Needless multiplication...

/sbin/atacontrol
/sbin/camcontrol
/sbin/comcontrol
/sbin/ifconfig
/sbin/ldconfig
/sbin/sconfig
/usr/sbin/acpiconf
/usr/sbin/arlcontrol
/usr/sbin/nxtconfig
/usr/sbin/fwcontrol
/usr/sbin/kbdcontrol
/usr/sbin/lptcontrol
/usr/sbin/mlxcontrol
/usr/sbin/raycontrol
/usr/sbin/sdpcontrol
/usr/sbin/vidcontrol
/usr/sbin/wicontrol

/sbin/atmconfig
/sbin/ccdconfig
/sbin/conscontrol
/sbin/kldconfig
/sbin/mdconfig
/sbin/spppcontrol
/usr/sbin/ancontrol
/usr/sbin/cdcontrol
/usr/sbin/fdcontrol
/usr/sbin/hccontrol
/usr/sbin/l2control
/usr/sbin/memcontrol
/usr/sbin/pciconf
/usr/sbin/rndc-confgen
/usr/sbin/sicontrol
/usr/sbin/vnconfig
/usr/sbin/wlconfig

This is not the errno you look for.

- For system calls which can only do simple thing, simple error categories are fine.

- For configuring TCP/IP over CLAW on an ESCON fiber in the precense of Escon directors ”EINVAL” will just not do.

Ioctl's other weakness.

```
# fooctrl -mode bidir -c1,3-8 -l21 -f foo.conf
fooctrl: Invalid Argument
# fooctrl -mode duplex -c1,3-8 -l21 -f foo.conf
fooctrl: Invalid Argument
# fooctrl -mode auto -c1,3-8 -l21 -f foo.conf
fooctrl: Invalid Argument
# fooctrl -h
fooctrl: Usage: fooctrl -mode <mode> <args>...
# /usr/games/fortune
To err is human -- to blame it on a computer is
even more so.
# ^D
```

Workarounds

- Private errno in the struct passed in ioctl.
- Separate ioctl: ”retrieve last error”.
- Return line number of failed test.
- Print cause message on console.
- Log cause message to logfile
- Break combo-operation into tens of steps.
- etc.

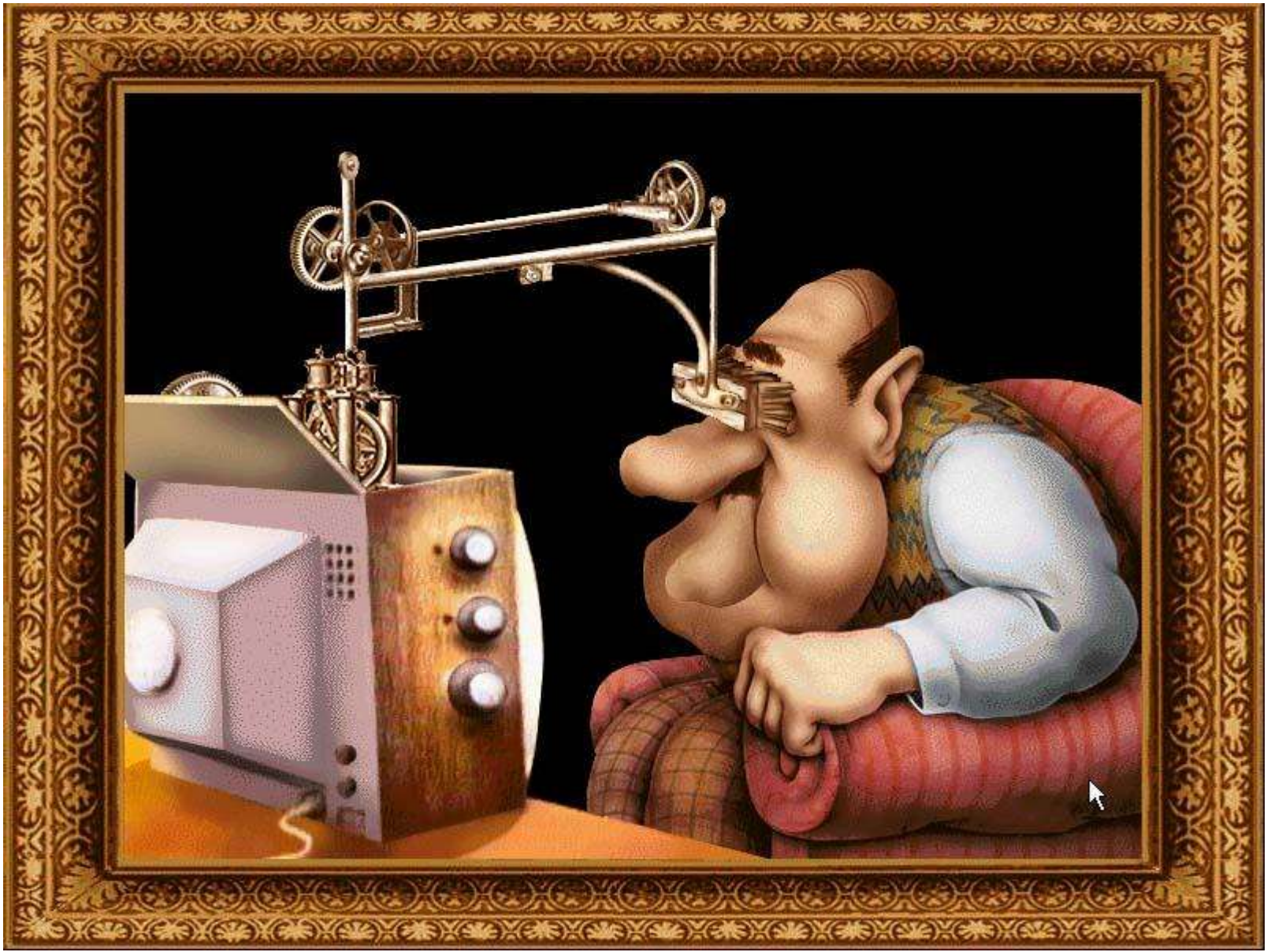
In-band / out-of-band.

- In-band:
    - Move the tape one file forward.
  - Addressing is implicit (file handle)
- Out-of-band:
    - Rescan SCSI bus for new devices.
  - Addressing is explicit.

In-band / out-of-band.

- Makes a BIG difference security wise.
- Using in-band for out-of-band is bad:
  - ”Eject that other CDROM”
- Using out-of-band for in-band has issues
  - ”Rewind that tape”
- Ioctl(2) is in-band
- Sysctl(2) is out-of-band.

And now for something entirely different...



What if there is no device driver ?

- Ioctl(2) needs a file descriptor.
- What if we don't have a device driver ?
- Non-device administrative interfaces:
  - Mount
  - Sysctl
  - Other (make a device driver anyway!)

mount(2)

- Different filesystems needs different parameters.
- Some parameters are shared
  - R/O vs R/W
  - NOEXEC, NODEV, NOSUID etc.
- Mount(2) passes a pointer to fs-private stuff.

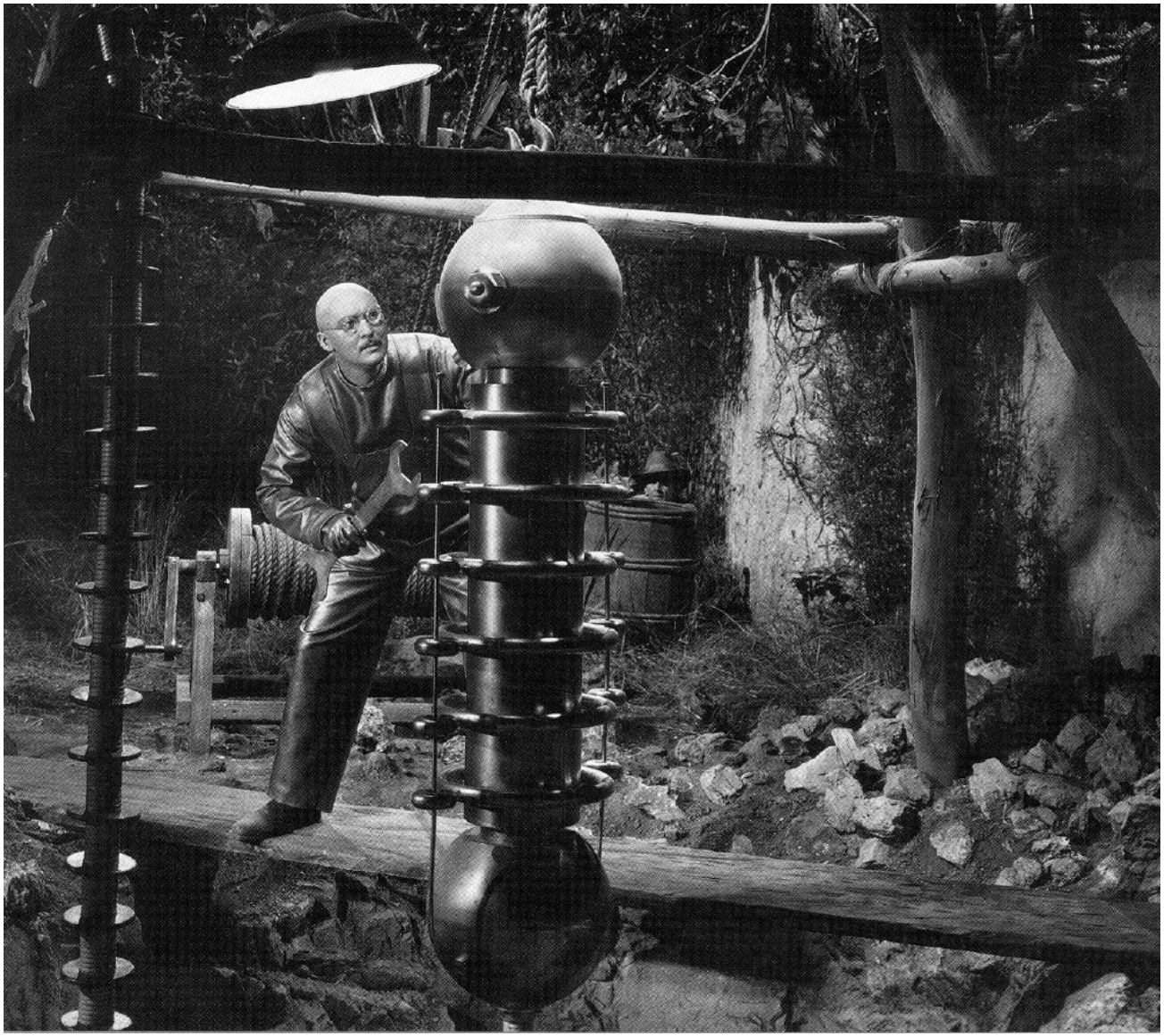
The exact same mess!

- Each filesystem needs a specialized userland program:
  - mount_ufs, mount_msdosfs, mount_cd9660...
- Insufficient parameter checking.
- ABI instability every time filesystem grows an option.
- (even worse than ioctl actually: no 'request' argument available).

Sysctl – a hack.

- Only structure imposed is namespace.
- Very flexible and easy to use.
- Generally not documented.
- Moves a variable length byte sequence in/out of the kernel.
- Clean in source, ugly in implementation.
- The real kitchensink.

Meanwhile in the lab...



GEOM

- GEOM is a framework.
- Methods are plugged in as needed.
- Methods can do anything:
  - RAID-0,1,3,5,10
  - Paritioning
  - Ship requests to userland
  - Encryption

GEOM OaM interface.

- The old way:
  - Each class defines ioctls.
  - Each class has fooctrl(8) program.
- Nothing works together.
- A unified administrator tool is not feasible.
  - Per class loadable objects for mgt program ?

Unified OaM

- Export global state of GEOM
  - Extensible format (XML)
    - (Different issue, not discussed here).
- Define API for sending instructions to GEOM classes and instances.
  - Without need for encoding instructions.

What is it we really need ?

- We need a conduit for passing commands from userland to some code in the kernel.
- The command consists of
  - Address
    - What piece of code in the kernel.
  - Verb
    - What action
  - Parameters

”Parameters”

- Kernel has extensible subsystems.
  - NetGraph, GEOM, device drivers, KLDS.
- Size and Format must be flexible.
  - Must be able to cater for all.
- Format unknown at compile time.
  - At least in the userland/foocontrol() end.

Extensible & Variable formats

- Encoded
  - Needs code to parse and encode user input
    - XML
    - Netgraph Parse
    - ASN.1
- Direct
  - Pass user input directly as text.
    - argc/env/config file.

Abstract notations

- ASN.1
  - ”This is not the format you are looking for.”
- Netgraph parse code.
  - Convert to byte stream.
  - Metadata in boths ends to control conversion.
- XML
  - Theoretically perfect
  - Practically overkill.

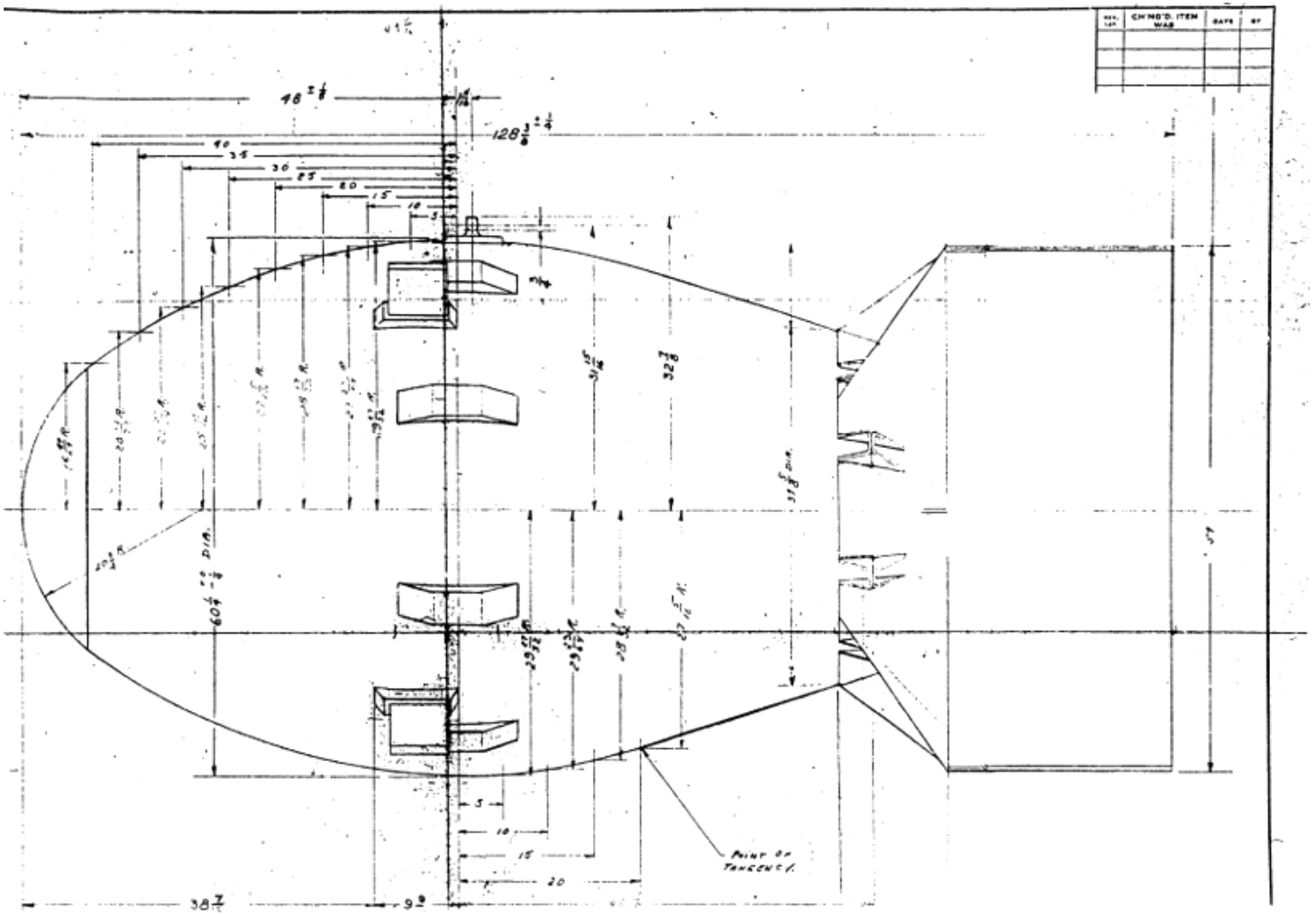
Direct transfer

- ”Communicate, don't interpret”
- Userland passes string input to kernel.
- Kernel does parsing & validation.
- Advantage:
  - No per class userland code
- Disadvantage:
  - Parsing strings in the kernel.

Lets kill a stigma...

- Parsing and validating strings into information is not banned in the kernel.
- Doesn't take more code than parsing and validating a binary format.
- ... or a encoded structure with multiple historical versions.

A new design...



The G_ctl API

- Build request ”environment style”
- Add elements as required.
- Issue request
- Check for errors.

G_ctl Example

```
struct gctl_req *r;
const char **errstr;

r = gctl_get_handle();
gctl_ro_param(r, "verb", -1, "create geom");
gctl_ro_param(r, "class", -1, "BDE");
gctl_ro_param(r, "provider", -1, dest);
errstr = gctl_issue(r);
if (errstr != NULL)
    errx(1, "Attach to %s failed: %s",
        dest, errstr);
```

G_ctl Example

```
# geom verb=”create geom” class=BDE provider=$d
```

Important points...

- Each element has Read/Write status:
  - Read-only: gctl_ro_param()
  - Read-write: gctl_rw_param()
    - Must specify buffer size
- First error message is latched.
  - All subsequent calls become no-ops.

Nmount API

- Same general principle:
  - fstype=msdos
  - fsname=/dev/da0a
  - fspath=/mnt
- Trickier:
  - Backwards compatible semantics necessary.
- Different implementation than g_ctl.

Conclusions (sort of)

- g_ctl and nmount breaks new ground.
- Much other code has similar needs:
  - Arguments to loadable device drivers
  - Sysctl variables controlling code.
  - Ifconfig(8) and network interfaces.
- Should we generalize to cover all ?