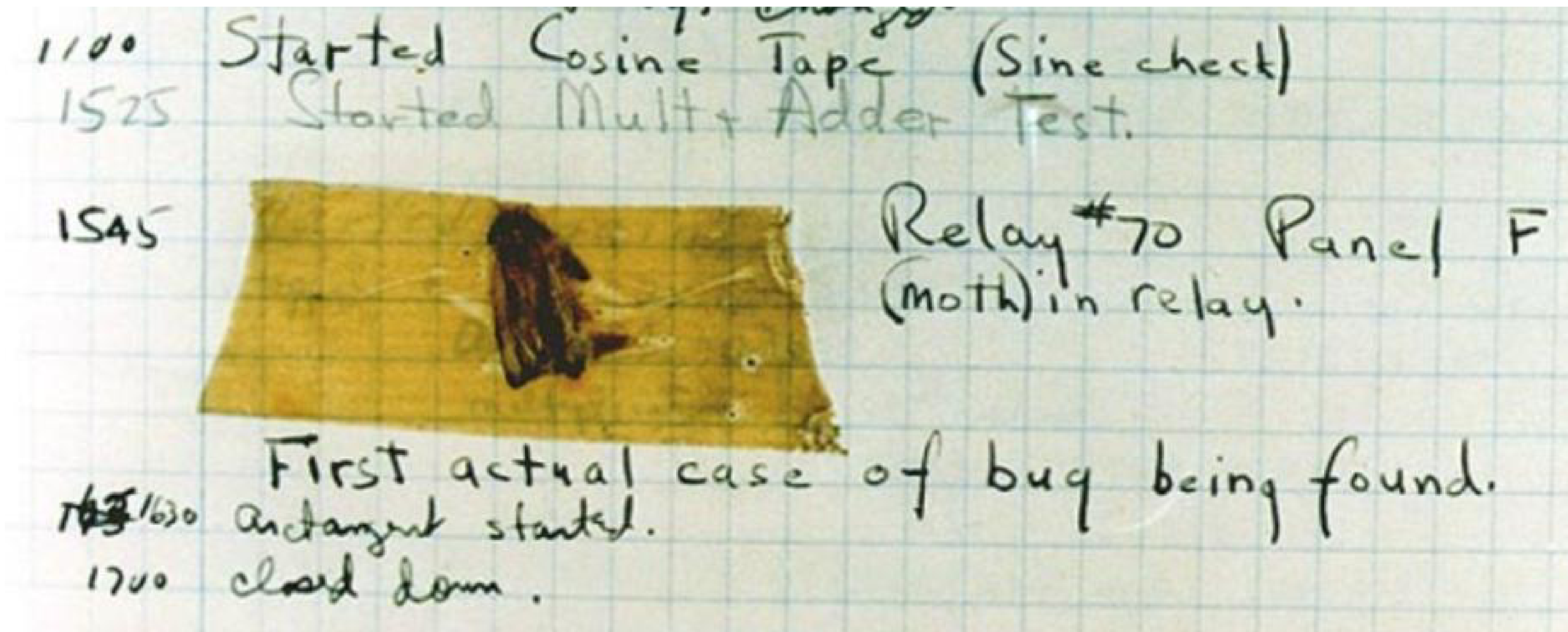
Debugging

Software Bugs

Every program has at least one bug

- If you remove your bug you still have a program
- And by definition it has at least one bug

Bugs grow with the square of the number of code lines



Hardware Bugs

Even hardware designs have bugs

- Two signals are swapped
- A power rail is not connected
- You forgot to add a pull-up

The most failure-prone part of PCB design is power-on and off

- You have transient voltage levels on both power and signals

And integrated circuits have bugs too

- The good thing here is that they usually are documented
  - Please read the errata sheet from page 1 to page end

Actually, every design features at least one bug

- The problem with sotware is that you think you can fix later

Different Debugging Approaches

You can remove a bug by patching code or hardware

- With your editor and compiler
- With the cutter and the soldering iron

You can work around a bug without removing it

- Higher-level software layers can bypass lower-level misfeatures
- Higher level hardware (PCB) can overcome lower-level (IC) bugs
- A software procedure can deal with hardware misbehavior

Or you can promote a bug to a feature

- And this is definitely the winning technique

No joking: it works. Everybody does it, try yourself.

- "If X happens the result is undefined"
- "The library does not support Y"
- "Errors are not reported"
- This also means you can simplify the code

"Debugging"

We call "debugging" the activity of identifying bugs

- This usually means inspecting code and data at runtime

We call "debugger" sth that stops the program so we can look at its guts

- We have a dead corpse to look at, but the world goes on
- In a concurrent environment this usually doesn't work
- And even your uC system is concurrent with the physical events you measure

We can debug by logging

- Don't be shy of printf, it's more powerful than a "debugger"
- Collect your logs, and make your graphs

But remember that diagnostics has a cost

- A serial port at 115200 takes 1ms to deliver 11 bytes
- A USB connection has more throughput, but some code and time overhead

Using the Oscilloscope

By moving GPIO pins, you can look at how the system behaves, especially when the situation is a repeated often

The overhead you add is usually very limited

- One machine instruction, a few clock cycles
- But sometimes GPIO operations are slow, because of clock domains

Measuring jitter, average and standard deviation is difficult

- The information is neither persistent nor complete

Measuring the worst case is possible

- Just trigger on a pulse "larger than XX" until it stops triggering

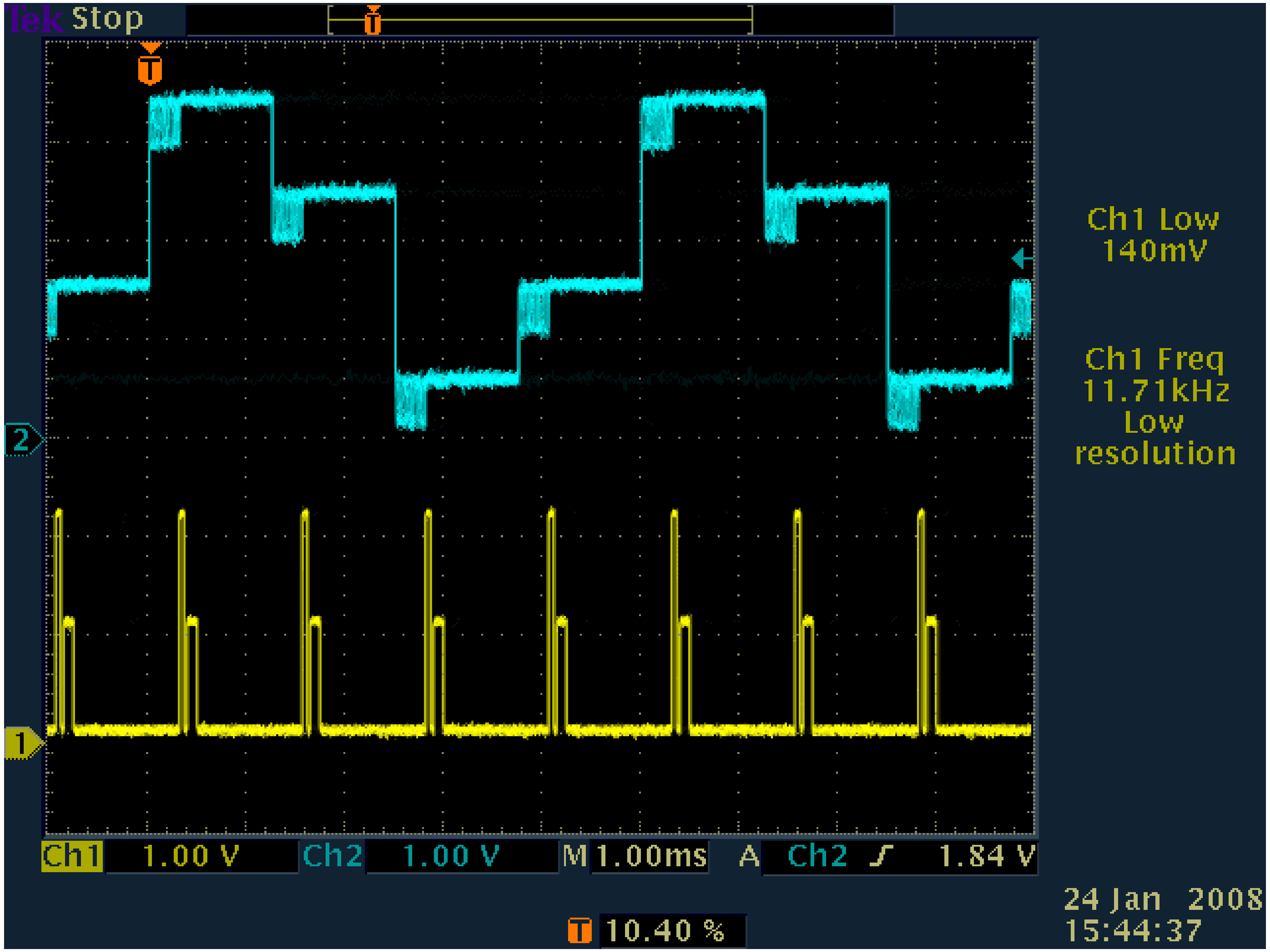
It is possible, though, to look at several pins per probe

- You only need a resistor network
- You can easily look at up to 4-5 pins per probe

An example scope run

The following figure shows a dot printer

- Blue: two phases of a stepper motor and the SPI clock of a data transfer
- Yellow: two "heather" pulses



The debugger: gdb

GDB (the GNU) debugger is a complex tool

- It includes a language interpreter
- It manages a number of file formats (even within ELF)
- It has scripting capabilities

But it is text based

- You can choose your preferred front-end
- Or none at all

Most important gdb commands

- list (list source at current point)
- bt (back trace the stack)
- info registers (show registers)
- disass
- break
- step, stepi
- next until
- p <expression to print>

The Mechanism: Ptrace

ptrace(2) is how a debugger can control a process

```
long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

- PTRACE_TRACEME
- PTRACE_ATTACH
- PTRACE_KILL
- PTRACE_DETACH

- TRACE_PEEKTEXT, PTRACE_PEEKDATA
- PTRACE_PEEKUSER
- PTRACE_POKETEXT, PTRACE_POKEDATA
- PTRACE_POKEUSER
- PTRACE_GETREGS, PTRACE_GETFPREGS
- PTRACE_SETREGS, PTRACE_SETFPREGS
- PTRACE_CONT
- PTRACE_SYSCALL, PTRACE_SINGLESTEP

Example: ptracetest

Tracing system calls: "strace"

"strace" is a very good tool for diagnosing programs

- To look for configuration files
- To see how child processes are called
- To understand how a network connection is made
- To diagnose an unclear error message
- To see where the program is spending its time

Strace is just a user of ptrace(2)

Examples:

```
strace -f -e execve -p `pidof inetd`
strace -f -e open,stat /etc/init.d/dhcp start
strace -tf -o /tmp/trace application args
```

Using gdbserver

gdbserver is the remotization of ptrace(2)

- You run gdbserver on the target like you'd invoke gdb
- You control gdbserver through a serial or TCP port

On the host, a specific target-aware gdb is used

- It must read and understand information in the target ELF
- It must be able to disassemble for the target architecture

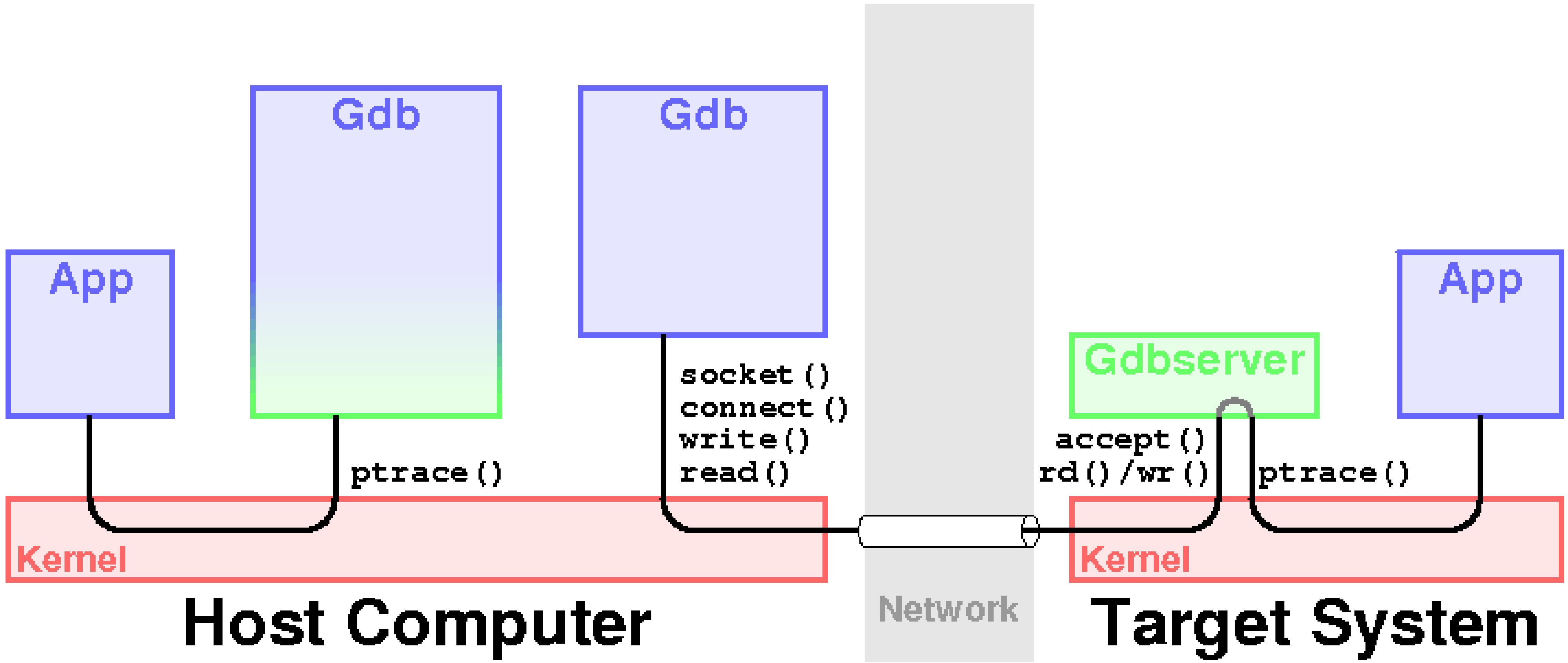
The cross-gdb (e.g.: arm-linux-gdb) si called just like gdb

- It uses a binary file with all symbolic information
- It accepts the "target remote" command to export ptrace actions

```
$ gdbserver host:2000  myapp
$ arm-linux-gdb  vmlinux
    target remote peedi:2000
    set print elements 16384
    set height 30
    p log_buf
    bt
```

How gdb and gdbserver work

Gdbserver can be considered a detached part of gdb



JTAG Debuggers

JTAG: Joint Test Access Group (like JPEG)

It was born as a pcb-testing tool

- You could read and write individual pins

It is now mainly a sw debugging tool

- You can read and write internal bits as well
  - You can access machine registers
  - You can send commands and read results
- Clearly, you can still drive pins
  - You can thus drive external hardware
  - e.g.you can program your external flash/eeprom

The easiest use case is running a host application

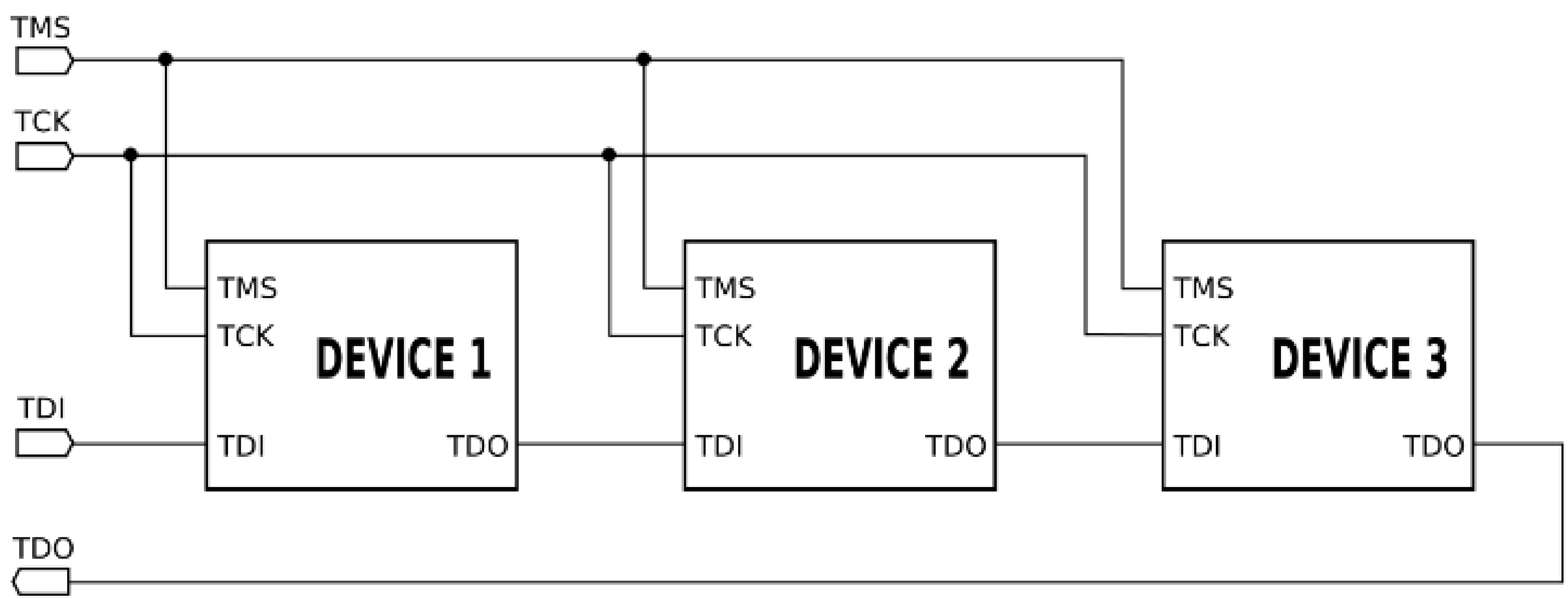
- The application talks the gdbserver protocol
- ... and drives jtag signals using an external tool

OpenOCD works in this way

JTAG is Just a Shift Register

JTAG is as simple as possible

- Unfortunately, it benefits from higher speeds



The cheapest jtag adapters used to run on the parallel port

- But this was horribly slow

A interesting alternative is using an FTDI chip

- They are usb adapters for uart and GPIO
- Most cheap commercial tools are likely based on these

Bigger JTAG tools

You can get bigger standalone JTAG devices

- You connect using Ethernet talking the gdbserver protocol
- They usually have they own protocol as well (different TCP port)
- Serious companies document all protocols

Unfortunately, they cost real money

- Example:

The GDB Stub

The simplest approach is using a GDB stub

- It can use TCP or UDP or a serial port
- You find a working one in the hsw2020 repository
- It costed 4 hours studying and 6 hours coding.

You need a way to interrupt your processor

- So your channel must be interrupt-driven (I used UART)
- You need to save all of your status (we do not, currently)
  - We do not need all registers for scheduling, so we don't save them

And you need a way to read and write memory

- The debugger examines code and data (code is read-only)

Debugging Support in Hardware

In addition to JTAG (external control)
the processor usually offers a debug register block

This is a set of registers that the CPU itself can use

- Single-step control
  - If set, returning from interrupt executes one instruction only
- Breakpoint registers
  - Usually the debugger writes to code space
  - But in the uC world it's not possible, you need hw support
- Watchpoint registers (write and/or read)
  - Doing this in software is awfully slow

The Cortex-M architecture specifies an optional debug unit

- And the 11U family does not implement it
- So no breakpoints, ever.

Unless we wisely instrument the code

- Any ideas about how to do it?

The Best Debugging is Not Debugging

If your code is bugless, you win

Sometimes, debugging tools are a loose

- You get sloppy in developing because you trust the tools
- You tend to enter "trial and error" mode
- You waste a lot of time in chasing stupid bugs

Linus Torvalds refused for years to have a GDB stub in Linux

I personally abused JTAG, and repented

Be eager, be cool