14:07 Routing Ethernet over GDB and SWD for Glitching

by Micah Elizabeth Scott

Hello again friendly and distinguished neighbors! As you can see, I've already started complimenting you, in part to distract from the tiny horrors ahead. Lately I've been spending some time experimenting on chips, injecting faults, and generally trying to guess how they are programmed. The results are a delightful topic that we have visited some in the past, and I'll surely weave some new stories about my results in the brighter days to come. For now, deep in the thick of things, you see, the glitching is monotonous work. Today's article is a tidbit about one particular solution to a problem I found while experimenting with voltage glitching a network-connected microcontroller.

Problem with Time Bubbles

Slow experiments repeat for days, and the experiments are often made slower on purpose by underclocking, broadening the little glitch targets we hope to peck at in order for the chip to release new secrets. To whatever extent I can, I like to control the clock frequency of a device under investigation. It helps to vary at least one clock to understand which parts of the system are driven by which clock sources. A slower clock can reduce the complexity of the tools you need for power analysis, accurate fault injection, and bus tracing.

If we had a system with a fully static design and a single clock, there wouldn't be any limit to the underclocking, and the system would follow the same execution path even if individual clock edges were delivered bi-weekly by pigeon. In reality, systems usually have additional clock domains driven by free-running oscillators or phase-locked loops (PLLs). This system design can impose limits on the practical amount of underclock you can achieve before the PLL fails to lock, or a watchdog timer expires before the software can make sufficient progress. On the bright side, these individual limitations can themselves reveal interesting information about the system's construction, and it may even be possible to introduce timing-related glitches intentionally by varying the clock speed.

These experiments create a bubble of alternate time, warped to your experiment's advantage. Any protocol that traverses the boundary between underclocked and real-time domains may need to be modified to account for the time difference. An SPI peripheral easily accepts a range of SCLK frequencies, but a serial port expecting 115,200 baud will have to know it's getting 25,920 baud instead. Most serial peripherals can handle this perfectly acceptably, but you may notice that operating systems and programming APIs start to turn their nose up at such a strange bit rate. Things become even less convenient with fixed-rate protocols like USB and Ethernet.

As fun as it would be to implement a custom Ethernet PHY that supports arbitrary clock scaling, it's usually more practical to extend the time bubble, slowing the input clock presented to an otherwise mundane Ethernet controller. For this technique to work, the peripheral needs a flexible interfacing clock. A USB-to-Ethernet bridge like the one on-board a Raspberry Pi could be underclocked, but then it couldn't speak with the USB host controller. PCI Express would have a similar problem.

SPI peripherals are handy for this purpose. My earlier Facewhisperer mashup of Facedancer and ChipWhisperer spoke underclocked USB by including a MAX3421E chip in the victim device's time domain. This can successfully break free from the time bubble, thanks to this chip talking over an SPI interface that can run at a flexible rate relative to the USB clock.

At first I tried to apply this same technique to Ethernet, using the ENC28J60, a 10baseT Ethernet controller that speaks SPI. This is even particularly easy to set up in tandem with a (non-underclocked) Raspberry Pi, thanks to some handy device tree overlays. This worked to a point, but the ENC28J60 proved to be less underclockable than my target microcontroller.

There aren't many SPI Ethernet controllers to choose from. I only know of the '28J60 from Microchip and its newer siblings with 100baseT support. In this case, it was inconvenient that I was dealing with two very different internal PHY designs on each side of the now very out-of-spec Ethernet link. I started making electrical changes, such as removing the AC coupling transformers, which needed somewhat different kludges for each type of PHY. This was getting frustrating, and seemed to be limiting the consistency of detecting a link successfully at such weird clock rates.

At this point, it seemed like it would be awfully convenient if I could just use the exact same kind of PHY on both sides of the link. I could have rewritten my glitch experiment request generator program as a firmware for the same type of microcontroller, but I preferred to keep the test code written in Python on a roomy computer so I could prototype changes quickly. These constraints pointed toward a fun approach that I had not seen anyone try before.

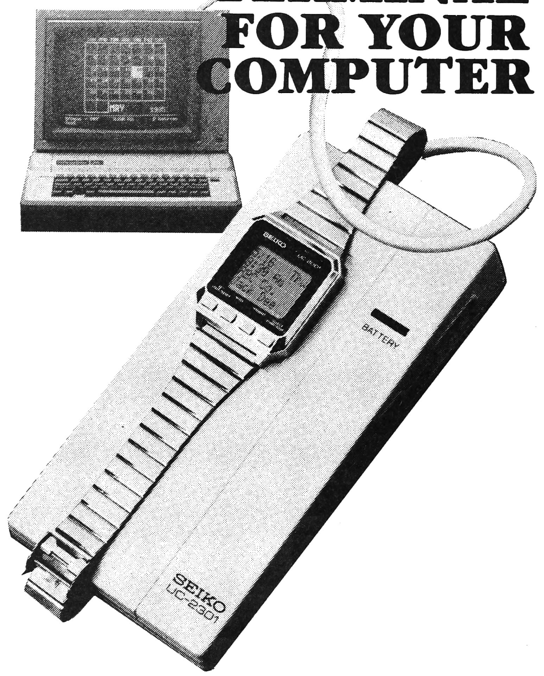
Ethernet over GDB

When I'm designing anything, but especially when I'm prototyping, I get a bit alarmed any time the design appears to have too many degrees of freedom. It usually means I could trade some of those extra freedoms for the constraints offered by an existing component somehow, and save from reinventing all the boring wheels.

The boring wheel I'd imagined here would have been a firmware image that perhaps implements a simple proxy that shuttles network frames and perhaps link status information between the on-chip Ethernet and an arbitrary SPI slave implementation. The biggest downside to this is that the SPI interface would have to speak another custom protocol, with yet another chunk of code necessary to bridge that SPI interface to something usable like a Linux network tap. It's tempting to implement standard USB networking, but an integrated USB controller would ultimately use the same clock source as the Ethernet PHY. It's tempting to emulate the ENC28J60's SPI protocol to use its existing Linux driver, but emulating this protocol's quick turnaround between address and data without getting an FPGA involved seemed unlikely.

In this case, the microcontroller hardware was already well-equipped to shuttle data between its on-chip Ethernet MAC and a list of packet buffers in main RAM. I eventually want a network device in Linux that I can really hang out with, capturing packets and setting up bridges and all. So, in the interest of eliminating as much glue as possible, I should be talking to the MAC from some code that's also capable of creating a Linux network tap.

GET A WRIST TERMINAL FOR YOUR COMPUTER



The Seiko Datagraph UNDER $200

Available for:

Apple II, II +, IIc, IIe
IBM-PC and compatibles
Commodore TRS-80

FOR COMPLETE PRODUCT LIST AND INFORMATION SEND ONE DOLLAR TO:

HI-TECH RECORD SYSTEMS©

13424 Whittier Blvd
Whittier, CA 90605

or call
In L.A. **(213) 945-2668**
Outside L.A. **(800) 448-5709**

ORDER NOW FOR CHRISTMAS DELIVERY

```
int main(void){
  MAP_SysCtlMOSCConfigSet(SYSCTL_MOSC_HIGHFREQ);
  g_ui32SysClock = MAP_SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
                SYSCTL_OSC_MAIN |
                SYSCTL_USE_PLL |
                SYSCTL_CFG_VCO_480), 120000000);

  PinoutSet(true, false);

  MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_EMAC0);
  MAP_SysCtlPeripheralReset(SYSCTL_PERIPH_EMAC0);
  MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_EPHY0);
  MAP_SysCtlPeripheralReset(SYSCTL_PERIPH_EPHY0);
  while (!MAP_SysCtlPeripheralReady(SYSCTL_PERIPH_EMAC0));

  MAP_EMACPHYConfigSet(EMAC0_BASE,
               EMAC_PHY_TYPE_INTERNAL |
               EMAC_PHY_INT_MDI_SWAP |
               EMAC_PHY_INT_FAST_L_UP_DETECT |
               EMAC_PHY_INT_EXT_FULL_DUPLEX |
               EMAC_PHY_FORCE_10B_T_FULL_DUPLEX);

  MAP_EMACReset(EMAC0_BASE);

  MAP_EMACInit(EMAC0_BASE, g_ui32SysClock,
           EMAC_BCONFIG_MIXED_BURST | EMAC_BCONFIG_PRIORITY_FIXED,
           8, 8, 0);

  MAP_EMACConfigSet(EMAC0_BASE,
          (EMAC_CONFIG_FULL_DUPLEX |
           EMAC_CONFIG_7BYTE_PREAMBLE |
           EMAC_CONFIG_IF_GAP_96BITS |
           EMAC_CONFIG_USE_MACADDR0 |
           EMAC_CONFIG_SA_FROM_DESCRIPTOR |
           EMAC_CONFIG_BO_LIMIT_1024),
          (EMAC_MODE_RX_STORE_FORWARD |
           EMAC_MODE_TX_STORE_FORWARD ), 0);

  MAP_EMACFrameFilterSet(EMAC0_BASE, EMAC_FRMFILTER_RX_ALL);

  init_dma_frames();

  MAP_EMACTxEnable(EMAC0_BASE);
  MAP_EMACRxEnable(EMAC0_BASE);

  while (1) {
    capture_phy_regs();
    __asm__ volatile ("bkpt");
  }
}
```

Figure 5. TM4C129x Firmware

This is where GDB, OpenOCD, and the Raspberry Pi really save the day. I thought I was going to be bit-banging the Serial Wire Debug (SWD) protocol again on some microcontroller, then building up from there all of the device-specific goodies necessary to access the memory and peripheral bus, set up the system clocks, and finally do some actual internetworking. It involves a lot of tedious reimplementation of things the semiconductor vendor already has working in a different language or a different format. But with GDB, we can make a minimal Ethernet setup firmware with whatever libraries we like, let it initialize the hardware, then inspect the symbols we need at runtime to handle packets.

At this point I can already hear some of you groaning about how slow this must be. While this debug bus won't be smoking the tires on a 100baseT switch any time soon, it's certainly usable for experimentation. In the specific setup I'll be talking about in more detail below, the bit-bang SWD bus runs at about 10 megabits per second peak, which turns into an actual sustained Ethernet throughput of around 130 kilobytes per second. It's faster than many internet connections I've had, and for microcontroller work it's been more than enough.

There's a trick to how this crazy network driver is able to run at such blazingly adequate speeds. Odds are if you're used to slow on-chip debugging, most of the delays have been due to slow round trips in your communication with the debug adapter. How bad this is depends on how low-level your debug adapter protocol happens to be. Does it make you schedule a USB transfer for every debug transaction? There goes a millisecond. Some adapters are much worse, some are a little better. Thanks to the Raspberry Pi 2 and 3 with their fast CPU and memory-mapped GPIOs, an OpenOCD process in userspace can bitbang SWD at rates competitive with a standalone debug adapter. By eliminating the chunky USB latencies we can hold conversations between hardware and Python code impressively fast. Idle times between SWD transfers are 10-50 microseconds when we're staying within OpenOCD, and as low as 150μs when we journey all the way back to Python code.

After building up a working network interface, it's easy to go a little further to add debugging hooks specific to your situation. In my voltage glitching setup, I wanted some hardware to know in advance when it was about to get a specific packet. I could add some string matching code to the Python proxy, using the Pi's GPIOs to signal the results of categorizing packets of interest. This signal itself won't be synchronized with the Ethernet traffic, but it was perfect for use as context when generating synchronized triggers on a separate FPGA.

You're being awfully vague, I thought there was a proof of concept here?

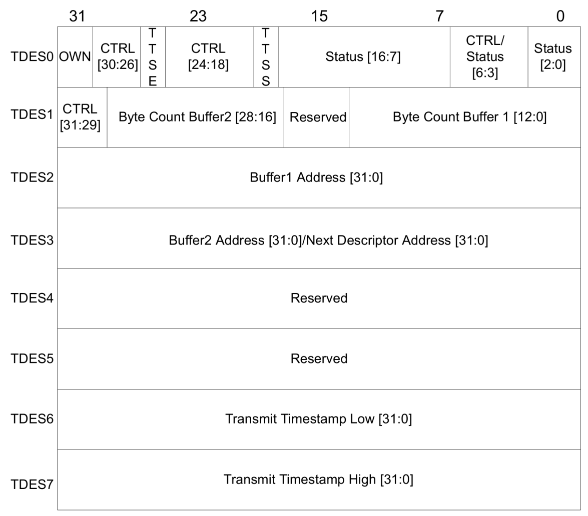
Okay, okay. Yes, I have one, and of course I'll share it here. But I did have a point; the whole process turned out to be a lot more generic than I expected, thanks to the functionality of OpenOCD and GDB. The actual code I wrote is very specific to the SoC I'm working with, but that's because it reads like a network driver split into a C and a Python portion.

If you're interested in a flexibly-clocked Ethernet adapter for your Raspberry Pi, or you're hacking at another network-connected device with the same micro, perhaps my code will interest you as-is, but ultimately I hope my humble PoC might inspire you to try a similar technique with other micros and peripherals.

Criterion
Stereopticons, Magic Lanterns & Accessories.
Perfect Apparatus for Visual Teaching
Scientific Projections & Private Use.
Electric Focusing Lamps
For Photo-Engravers, Stage Lighting, Etc.
Send for Catalog.
Agencies in Boston, Philadelphia, Atlanta, St. Louis, Buffalo, Chicago, San Francisco.
J.B. Colt & Co.,
115-117 Nassau St., New York.

Tiva GDBthernet

So the specific chip I've been working with is a 120 MHz ARM Cortex-M4F core with on-board Ethernet, the TM4C129x, otherwise known as the Tiva-C series from Texas Instruments. Luckily there's already a nice open source project to support building firmware for this platform with GCC.[17] The platform includes some networking examples based on the uIP and lwIP stacks. For our purposes, we need to dig a bit lower. The on-chip Ethernet MAC uses DMA both to transfer packet contents and to access a queue made from DMA Descriptor structures.

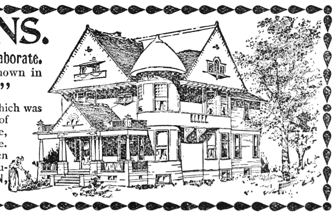


	31			23		15	7	0
TDES0	OWN	CTRL [30:26]	TTSE	CTRL [24:18]	TTSS	Status [16:7]	CTRL/ Status [6:3]	Status [2:0]
TDES1	CTRL [31:29]	Byte Count Buffer2 [28:16]			Reserved	Byte Count Buffer 1 [12:0]		
TDES2	Buffer1 Address [31:0]							
TDES3	Buffer2 Address [31:0]/Next Descriptor Address [31:0]							
TDES4	Reserved							
TDES5	Reserved							
TDES6	Transmit Timestamp Low [31:0]							
TDES7	Transmit Timestamp High [31:0]							

This data structure is convenient enough to access directly from Python when we're shuttling packets back and forth, but setting up the peripheral involves a boatload of magic numbers that I'd prefer not to fuss with. We can mostly reuse existing library code for this. The main firmware file `gdbthernet.c` uses a viscous wad of library calls to set up all the hardware we need, before getting itself stuck in a breakpoint loop, shown in Figure 5.

Everything in this file only needs to exist for convenience. The micro doesn't need any firmware whatsoever, we could set up everything from GDB. But it's easier to reuse whatever we can. You may have noticed the call to `capture_phy_regs()` above. We have only indirect access to the PHY registers via the Ethernet MAC, so it was a bit more convenient to reuse existing library code for reading those registers to determine the link state.

HOUSE PLANS.
All Grades; Cheap, Medium and Elaborate.
100 DESIGNS for DWELLINGS are shown in
"Artistic Dwellings,"
A book of house plans; the 1895 edition of which was issued in Feb'y last. These plans are the work of designers skilled in dwelling-house architecture, and embody the newest and latest ideas in this line. Views, floor plans, costs and descriptions with each design. PRICE, $1.00. Send stamp for circulars and sample designs. FRANK P. ALLEN, Architect, Grand Rapids, Michigan.



On the Raspberry Pi side, we start with a shell script `proxy.sh` that spawns an OpenOCD and GDB process, and tells GDB to run `gdb_net_host.py`. Some platform-specific configuration for OpenOCD tells it how to get to the processor and which micro we're dealing with. GDB provides quite high-level access to parse expressions in the target language, and the Python API wraps those results nicely in data structures that mimic the native language types. My current approach has been to use this parsing sparingly, though, since it seems to leak memory. Early on in `gdb_net_host.py`, we scrape all the constants we'll be needing from the firmware's debug symbols. (Figure 6.)

From here on, we'll expect to chug through all of the Raspberry Pi CPU cycles we can. There's no interrupt signaling back to the debugger, everything has to be based on polling. We could poll for Ethernet interrupts, but it's more expedient to poll the DMA Descriptor directly, since that's the data we actually want. Here's how we receive Ethernet frames and forward them to our tap device. (Figure 7.)

The transmit side is similar, but it's driven by the availability of a packet on the tap interface. You can see the hooks for GPIO trigger outputs in Figure 8.

That's just about all it takes to implement a pretty okay network interface for the Raspberry Pi. Attached you'll find the few necessary but boring tidbits I've left out above, like link state detection and debugger setup. I've been pretty happy with the results. This approach is even comparable in speed to the ENC28J60 driver, if you don't mind the astronomical CPU load. I hope this trick inspires you to create weird peripheral mashups using GDB and the Raspberry Pi. If you do, please be a good neighbor and consider documenting your experience for others. Happy hacking!

[17] `git clone https://github.com/yuvadm/tiva-c`

```
inf = gdb.selected_inferior()
num_rx = int(gdb.parse_and_eval('sizeof g_rxBuffer / sizeof g_rxBuffer[0]'))
num_tx = int(gdb.parse_and_eval('sizeof g_txBuffer / sizeof g_txBuffer[0]'))
g_phy_bmcr = int(gdb.parse_and_eval('(int)&g_phy.bmcr'))
g_phy_bmsr = int(gdb.parse_and_eval('(int)&g_phy.bmsr'))
g_phy_cfg1 = int(gdb.parse_and_eval('(int)&g_phy.cfg1'))
g_phy_sts = int(gdb.parse_and_eval('(int)&g_phy.sts'))
rx_status = [int(gdb.parse_and_eval(
        '(int)&g_rxBuffer[%d].desc.ui32CtrlStatus' % i)) for i in range(num_rx)]
rx_frame = [int(gdb.parse_and_eval(
        '(int)g_rxBuffer[%d].frame' % i)) for i in range(num_rx)]
tx_status = [int(gdb.parse_and_eval(
        '(int)&g_txBuffer[%d].desc.ui32CtrlStatus' % i)) for i in range(num_tx)]
tx_count = [int(gdb.parse_and_eval(
        '(int)&g_txBuffer[%d].desc.ui32Count' % i)) for i in range(num_tx)]
tx_frame = [int(gdb.parse_and_eval('(int)g_txBuffer[%d].frame' % i)) for i in range(num_tx)]
```

Figure 6. Fetching Debug Symbols

```
next_rx = 0

def rx_poll_demand():
    # Rx Poll Demand (wake up MAC if it's suspended)
    inf.write_memory(0x400ECC08, struct.pack('<I', 0xFFFFFFFF))

def poll_rx(tap):
    global next_rx

    status = struct.unpack('<I', inf.read_memory(rx_status[next_rx], 4))[0]
    if status & (1 << 31):
        # Hardware still owns this buffer; try later
        return

    if status & (1 << 11):
        print('RX Overflow error')
    elif status & (1 << 12):
        print('RX Length error')
    elif status & (1 << 3):
        print('RX Receive error')
    elif status & (1 << 1):
        print('RX CRC error')
    elif (status & (1 << 8)) and (status & (1 << 9)):
        # Complete frame (first and last parts), strip 4-byte FCS
        length = ((status >> 16) & 0x3FFF) - 4
        frame = inf.read_memory(rx_frame[next_rx], length)
        if VERBOSE:
            print('RX %r' % binascii.b2a_hex(frame))
        tap.write(frame)
    else:
        print('RX unhandled status %08x' % status)

    # Return the buffer to hardware, advance to the next one
    inf.write_memory(rx_status[next_rx], struct.pack('<I', 0x80000000))
    next_rx = (next_rx + 1) % num_rx
    rx_poll_demand()
    return True
```

Figure 7. Ethernet Frame RX

```
next_tx = 0
tx_buffer_stuck_count = 0

def tx_poll_demand():
    # Tx Poll Demand (wake up MAC if it's suspended)
    inf.write_memory(0x400ECC04, struct.pack('<I', 0xFFFFFFFF))

def poll_tx(tap):
    global next_tx
    global tx_buffer_stuck_count

    status = struct.unpack('<I', inf.read_memory(tx_status[next_tx], 4))[0]
    if status & (1 << 31):
        print('TX waiting for buffer %d' % next_tx)
        tx_buffer_stuck_count += 1
        if tx_buffer_stuck_count > 5:
            gdb.execute('run')
        update_phy_status()
        tx_poll_demand()
        return

  tx_buffer_stuck_count = 0
    if not select.select([tap.fileno()], [], [], 0)[0]:
        return
    frame = tap.read(4096)

    match_low = TRIGGER and frame.find(TRIGGER_LOW) >= 0
    match_high = TRIGGER and frame.find(TRIGGER_HIGH) >= 0

    if VERBOSE:
        print('TX %r' % binascii.b2a_hex(frame))

    if match_low:
       if VERBOSE:
           print('-' * 60)
       GPIO.output(TRIGGER_PIN, GPIO.LOW)

    inf.write_memory(tx_frame[next_tx], frame)
    inf.write_memory(tx_count[next_tx], struct.pack('<I', len(frame)))
    inf.write_memory(tx_status[next_tx], struct.pack('<I',
        0x80000000 | # DES0_RX_CTRL_OWN
        0x20000000 | # DES0_TX_CTRL_LAST_SEG
        0x10000000 | # DES0_TX_CTRL_FIRST_SEG
        0x00100000)) # DES0_TX_CTRL_CHAINED
    next_tx = (next_tx + 1) % num_tx

    if match_high:
        GPIO.output(TRIGGER_PIN, GPIO.HIGH)
        if VERBOSE:
            print('+' * 60)

    tx_poll_demand()
    return True
```

Figure 8. Ethernet Frame TX