5 SWD Marionettes; or, The Internet of Unsuspecting Things

by Micah Elizabeth Scott

Greetings, neighbors! Let us today gather to celebrate the Internet of Things. We live in a world where nearly any appliance, pet, or snack food can talk to the Cloud, which sure is a disarming name for this random collection of computers we've managed to network together. I bring you a humble PoC today, with its origins in the even humbler networking connections between tiny chips.

5.1 Firmware? Where we're going, we don't need firmware.

I've always had a fascination with debugging interfaces. I first learned to program on systems with no viable debugger, but I would read magazines in the nineties with articles advertising elaborate and pricey emulator and in-circuit debugger systems. Decades go by, and I learn about JTAG, but it's hard to get excited about such a weird, wasteful, and under-standardized protocol. JTAG was designed for an era when economy of silicon area was critical, and it shows.

More years go by, and I learn about ARM's Serial Wire Debug (SWD) protocol. It's a tantalizing thing: two wires, clock and bidirectional data, give you complete access to the chip. You can read or write memory as if you were the CPU core, in fact concurrently while the CPU core is running. This is all you need to access the processor's I/O ports, its on-board serial ports, load programs into RAM or flash, single-step code, and anything else a debugger does. I took my first dive into SWD in order to develop an automated testing infrastructure for the Fadecandy LED controller project. There was much yak shaving, but the result was totally worthwhile.

More recently, Cortex-M0 microcontrollers have been showing up with prices and I/O features competitive with 8-bit microcontrollers. For example, the Freescale MKE04Z8VFK4 is less than a dollar even in single quantities, and there's a feature-rich development board available for $15. These micros are cheaper than many single-purpose chips, and they have all the peripherals you'd expect from an AVR or PIC micro. The dev board is even compatible with Arduino shields.

In light of this economy of scale, I'll even consider using a Cortex-M0 as a sort of I/O expander chip. This is pretty cool if you want to write microcontroller firmware, but what if you want something without local processing? You could write a sort of pass-through firmware, but that's extra complexity as well as extra timing uncertainty. The SWD port would be a handy way to have a simple remote-controlled set of ARM peripherals that you can drive from another processor.

Okay! So let's get to the point. SWD is neat, we want to do things with it. But, as is typical with ARM, the documentation and the protocols are fiercely layered. It leads to the kind of complexity that can make little sense from a software perspective, but might be more forgivable if you consider the underlying hardware architecture as a group of tiny little machines that all talk asynchronously.

The first few tiny machines are described in the 250-page ARM Debug Interface Architecture Specification ADIv5.0 to ADIv5.2 tome.[26] It becomes apparent that the tiny machines must be so tiny because of all the architectural flexibility the designers wanted to accommodate. To start with, there's the Debug Port (DP). The DP is the lower layer, closest to the physical link. There are different DPs for JTAG and Serial Wire Debug, but we only need to be concerned with SWD.

We can mostly ignore JTAG, except for the process of initially switching from JTAG to SWD on

[26] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0031c/index.html`

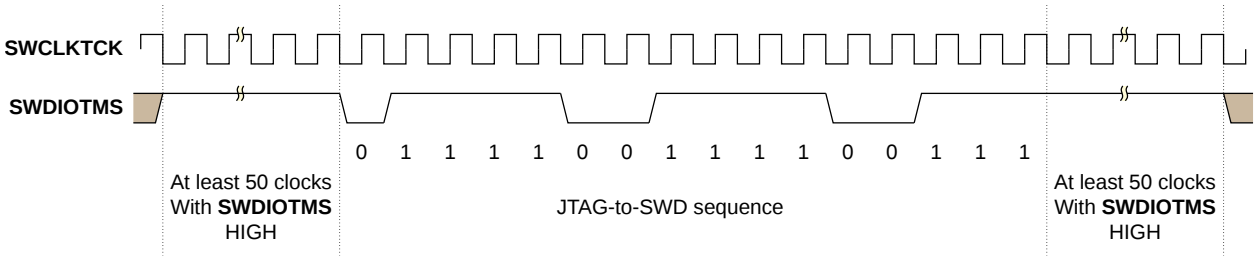

Figure 12 – JTAG-to-SWD sequence timing

systems that support both options. SWD's clock matches the JTAG clock line, and SWD's bidirectional data maps to JTAG's TMS signal. A magic bit sequence in JTAG mode on these two pins will trigger a switch to the SWD mode, as shown in Figure 12.

SWD will look a bit familiar if you've used SPI or I2C at all. It's more like SPI, in that it uses a fast and non-weird clocking scheme. Each processor's data sheet will tell you the maximum SWD speed, but it's usually upwards of 20 MHz. This hints at why the protocol includes so many asynchronous layers: the underlying hardware operates on separate clock domains, and the debug port may be operating much faster or slower than the CPU clock.

Whereas SPI typically uses separate wires for data in and out, SWD uses a single wire (it's in the name!) and relies on a "turnaround" period to switch bus directions during one otherwise wasted clock cycle that separates groups of written or returned bits. These bit groups are arranged into tiny packets with start bits and parity and such, using turnaround bits to separate the initial, data, and acknowledgment phases of the transfer. For example, see Figures 13 and 14 to execute read and write operations and for all the squiggly details on these packets, the tome has you covered starting with Figure 4-1.

These low-level SWD packets give you a memory-like interface for reading and writing registers; but we're still a few layers removed from the kind of registers that you'd see anywhere else in the ARM architecture. The DP itself has some registers accessed via these packets, or these reads and writes can refer to registers in the next layer: the Access Port (AP).

The AP could really be any sort of hardware that needs a dedicated debug interface on the SoC. There are usually vendor specific access ports, but usually you're talking to the standardized MEM-AP which gives you a port for accessing the ARM's AHB memory bus. This is what gives the debugger a view of memory from the CPU's point of view.

Each of these layers are of course asynchronous. The higher levels, MEM-AP and above, tend to have a handshaking scheme that looks much like any other memory mapped I/O operation. Write to a register, wait for a bit to clear, that sort of thing. The lower level communications between DP and AP needs to be more efficient, though, so reads are pipelined. When you issue a read, that transaction will be returning data for the previous read operation on that DP. You can give up the extra throughput in order to simplify the interface if you want, by explicitly reading the last result (without starting a new read) via a Read Buffer register in the DP.

This is where the Pandora's Box opens up. With the MEM-AP, this little serial port gives you full access to the CPU's memory. And as is the tradition of the ARM architecture, pretty much everything is memory-mapped. Even the CPU's registers are indirectly accessed via a memory mapped debug controller while the CPU is halted. Now everything in the thousands of pages of Cortex-M and vendor-specific documentation is up for grabs.

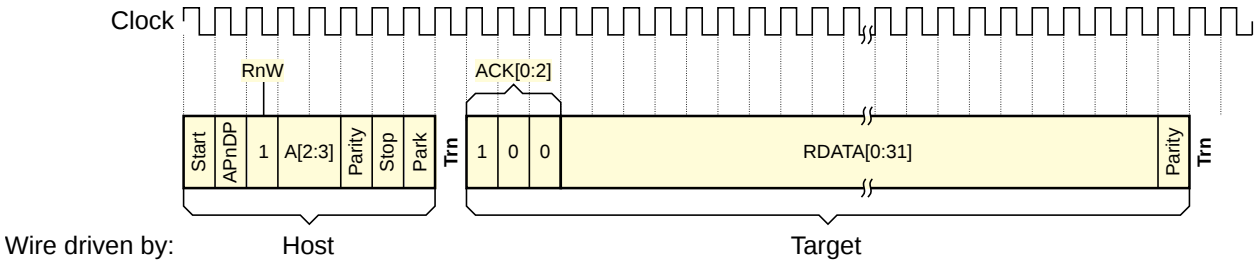

Figure 13 – Serial Wire Debug successful read operation

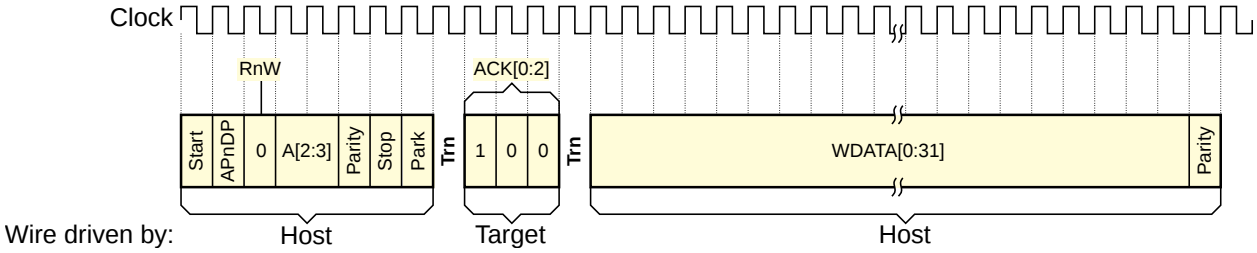

Figure 14 – Serial Wire Debug successful write operation

5.2 Now I'm getting to the point.

I like making tools, and this seems like finally the perfect layer to use as a foundation for something a bit more powerful and more explorable. Combining the simple SWD client library I'd written earlier with the excellent Arduino ESP8266 board support package, attached you'll find `esp8266-arm-swd`,[27] an Arduino sketch you can load on the $5 ESP8266 Wi-Fi microcontroller. There's a README with the specifics you'll need to connect it to any ARM processor and to your Wi-Fi. It provides an HTTP GET interface for reading and writing memory. Simple, joyful, and roughly equivalent security to most Internet Things.

These little HTTP requests to read and write memory happen quickly enough that we can build a live hex editor that continuously scans any visible memory for changes, and sends writes whenever any value is edited. By utilizing all sorts of delightful HTML5 modernity to do the UI entirely client-side, we can avoid overloading the lightweight web server on the ESP8266.

This all adds up to something that's I hope could

[27] `unzip pocorgtfo10.zip esp8266-arm-swd.zip`

```
<ul>
  <li>
    Turn the LED
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00100800"> red </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00200800"> green </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00300000"> blue </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00200000"> cyan </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00100000"> pink </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00000000"> whiteish </a>, or
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00300800"> off </a>
  </li>
  <li>
    Now <a is="swd-async-action" href="/api/halt"> halt the CPU </a> and let's have some scratch RAM:
    <p>
      <swd-hexedit addr="0x20000000" count="32"></swd-hexedit>
    </p>
  </li>
  <li>
    <a is="swd-async-action" href="/api/mem/write?0x20000000=0x22004b0a&.=0x4a0a601a&.=0x601a4b0a&.=0x4a0b4b0a&.=0x4b0b6013&.=0x2b003b01&.=0x2380d1fc&.=0x6013035b&.=0x3b014b07&.=0xd1fc2b00&.=0x46c0e7f0&.=0x40048008&.=0x00300800&.=0x400ff014&.=0x00200800&.=0x400ff000&.=0x00123456&.=0x7ffffbc&.=0x00000001">
      Load a small program
    </a>
    into the scratch RAM
  </li>
  <li>
    <a is="swd-async-action" href="/api/reg/write?0x3c=0x20000000"> Set the program counter </a>
    (<span is="swd-hexword" src="/api/reg" addr="0x3c"></span>)
    to the top of our program
  </li>
    <li>
      The PC <i>sample</i> register (<span is="swd-hexword" addr="0xe000101c"></span>)
      tells you where the <i>running</i> CPU is
    </li>
    <li>
      <a is="swd-async-action" href="/api/mem/write?0xE000EDF0=0xA05F0001"> Let the CPU run! </a>
      (or try a <a is="swd-async-action" href="/api/mem/write?0xE000EDF0=0xA05F0005"> single step </a>)
    </li>
    <li>
      While the program is running, you can modify its delay value:
      <span is="swd-hexword" addr="0x20000040"></span>
    </li>
</ul>
```

Figure 15 – Single Wire Debug from HTML5

be used for a kind of *literate* reverse engineering and debugging, in the way Knuth imagined literate programming. When trying to understand a new platform, the browser can become an ideal sandbox for both investigating and documenting the unknown hardware and software resources.

The included HTML5 web app, served by the Arduino sketch, uses some Javascript to define custom HTML elements that let you embed editable hex dumps directly into documentation. Since a register write is just an HTTP GET, hyperlinks can cause hardware state changes or upload small programs.

There's a small example of this approach on the "Memory Mapped I/O" page, designed for the $15 Freescale FRDM-KE04Z board. This one is handy as a prototyping platform, particularly since the I/O is 5V tolerant and compatible with Arduino shields. Figure 15 contains the HTML5 source for that demo.

This sample uses some custom HTML5 elements defined in `/script.js`: `swd-async-action`, `swd-hexedit`, and `swd-hexword`. The `swd-async--action` isn't so exciting, it's really just a special kind of hyperlink that shows a pass/fail result without navigating away from the page. The `swd-hexedit` is also relatively mundane; it's just a shell that expands into many `swd-hexword` elements. That's where the substance is. Any `swd--hexedit` element that's scrolled into view will be refreshed in a continuous round-robin cycle, and the content is editable by default. These become simple but powerful tools.

5.3 Put a chip in it!

While the practical applications of `esp8266-arm-swd` may be limited to education and research, I think it's an interesting Minimum Viable Internet Thing. With the ESP8266 costing only a few dollars, anything with an ARM microcontroller could become an Internet Thing with zero firmware modification, assuming you can find the memory addresses or hardware registers that control the parts you care about. Is it practical? Not really. Secure? Definitely not! But perhaps take a moment to consider whether it's really any worse than the other solutions at hand. Is ARM assembly and HTML5 your kind of fun? Please send pull requests. Happy hacking