# 17:06 The DIP Flip Whixr Trick: An Integrated Circuit That Functions in Either Orientation

*by Joe "Kingpin" Grand*

Hardware trickery comes in many shapes and sizes: implanting add-on hardware into a finished product, exfiltrating data through optical, thermal, or electromagnetic means, injecting malicious code into firmware, BIOS, or microcode, or embedding Trojans into physical silicon. Hackers, governments, and academics have been playing in this wide open field for quite some time and there's no sign of things slowing down.

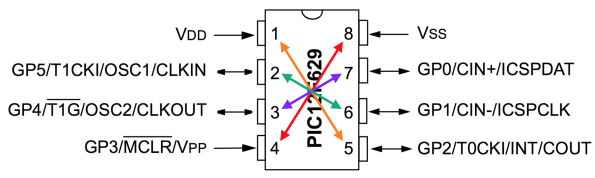
This PoC, inspired by my friend Whixr of `#tymkrs`, demonstrates the feasibility of an IC behaving differently depending on which way it's connected into the system. Common convention states that ICs must be inserted in their specified orientation, assisted by the notch or key on the device identifying pin 1, in order to function properly.

So, let's defy this convention!

– — — – — — — – — — –

Most standard chips, like digital logic devices and microcontrollers, place the power and ground connections at corners diagonal from each other. If one were to physically rotate the IC by 180 degrees, power from the board would connect to the ground pin of the chip or vice versa. This would typically result in damage to the chip, releasing the magic smoke that it needs to function. The key to this PoC was finding an IC with a more favorable pin configuration.

While searching through microcontroller data sheets, I came across the Microchip PIC12F629. This particular 8-pin device has power and GPIO (General Purpose I/O) pins in locations that would allow the chip to be rotated with minimal risk. Of course, this PoC could be applied to any chip with a suitable pin configuration.




In the pinout drawing, which shows the chip from above in its normal orientation, arrows denote the alternate functionality of that particular pin when the chip is rotated around. Since power (VDD) is normally connected to pin 1 and ground (VSS) is normally connected to pin 8, if the chip is rotated, GP2 (pin 5) and GP3 (pin 4) would connect to power and ground instead. By setting both GP2 and GP3 to inputs in firmware and connecting them to power and ground, respectively, on the board, the PIC will be properly powered regardless of orientation.

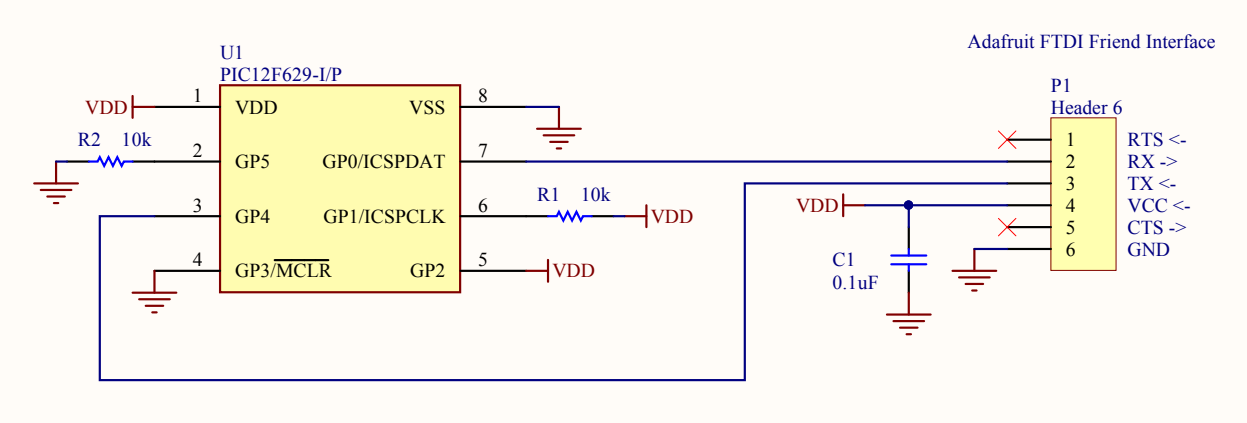
– — — – — — — – — — –

I thought it would be fun to change the data that the PIC sends to a host PC depending on its orientation.

On power-up of the PIC, GP1 is used to detect the orientation of the device and set the mode accordingly. If GP1 is high (caused by the pull-up resistor to VCC), the PIC will execute the normal code. If GP1 is low (caused by the pull-down resistor to VSS), the PIC will know that it has been rotated and will execute the alternate code. This orientation detection could also be done using GP5, but with inverted polarity.

The PIC's UART (asynchronous serial) output is bit-banged in firmware, so I'm able to reconfigure the GPIO pins used for TX and RX (GP0 and GP4) on-the-fly. The TX and RX pins connect directly to an Adafruit FTDI Friend, which is a standard FTDI FT232R-based USB-to-serial adapter. The FTDI Friend also provides 5V (VDD) to the PoC.

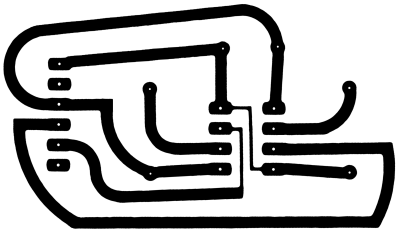
In normal operation, the device will look for a key press on GP4 from the FTDI Friend's TX pin and then repeatedly transmit the character 'A' at 9600 baud via GP0 to the FTDI Friend's RX pin. When the device is rotated 180 degrees, the device will look for a key press on GP0 and repeatedly transmit the character 'B' on GP4. As a key press detector, instead of reading a full character from the host, the device just looks for a high-to-low transition on the PIC's currently configured RX pin. Since that pin idles high, the start bit of any data sent from the FTDI Friend will be logic low.

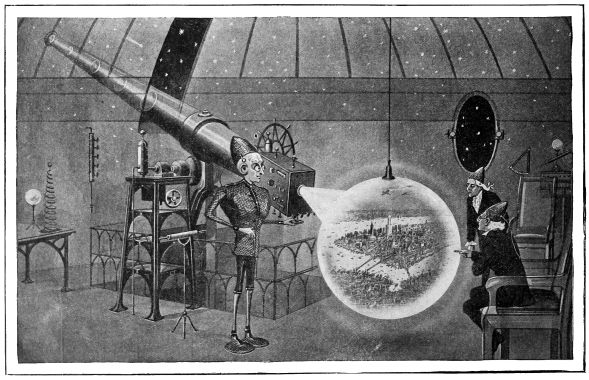



```
switch (input(PIN_A1)) {// orientation
    detection
  case MODE_NORMAL: // normal behavior
    #use rs232(baud=9600, bits=8, parity=N,
    stop=1, xmit=PIN_A0, force_sw)

    //wait for a keypress
    while(input(PIN_A4));

    while(1){
      printf("A ");
      delay_ms(10);
    }
    break;

  case MODE_ALTERNATE: // abnormal behavior
    #use rs232(baud=9600, bits=8, parity=N,
    stop=1, xmit=PIN_A4, force_sw)

    // wait for a keypress
    while(input(PIN_A0));

    while(1){
      printf("B ");
      delay_ms(10);
    }
    break;
}
```

For your viewing entertainment, a demonstration of my breadboard prototype can be found on Youtube.[17] Complete engineering documentation, including schematic, bill-of-materials, source code, and layout for a small circuit board module are also available.[18]

Let this PoC serve as a reminder that one should not take anything at face value. There are an endless number of ways that hardware, and the electronic components within a hardware system, can misbehave. Hopefully, this little trick will inspire future hardware mischief and/or the development of other sneaky circuits. If nothing else, you're at least armed with a snarky response for the next time some over-confident engineer insists ICs will only work in one direction!





[17] Joe Grand, Sneaky Circuit: This DIP Goes Both Ways

[18] `unzip pocorgtfo17.pdf dipflip.zip # or at www.grandideastudio.com/portfolio/sneaky-circuits/`

# 17:07 Injecting shared objects on FreeBSD with libhijack.

*by Shawn Webb*

In the land of red devils known as Beasties exists a system devoid of meaningful exploit mitigations. As we explore this vast land of opportunity, we will meet our ELFish friends, [p]tracing their very moves in order to hijack them. Since unprivileged process debugging is enabled by default on FreeBSD, we can abuse `ptrace` to create anonymous memory mappings, inject code into them, and overwrite PLT/-GOT entries.[19] We will revive a tool called libhijack to make our nefarious activities of hijacking ELFs via `ptrace` relatively easy.

Nothing presented here is technically new. However, this type of work has not been documented in this much detail, so here I am, tying it all into one cohesive work. In Phrack 56:7, Silvio Cesare taught us fellow ELF research enthusiasts how to hook the PLT/GOT.[20] Phrack 59:8, on Runtime Process Infection, briefly introduces the concept of injecting shared objects by injecting shellcode via `ptrace` that calls `dlopen()`.[21] No other piece of research, however, has discovered the joys of forcing the application to create anonymous memory mappings from which to inject code.

This is only part one of a series of planned articles that will follow libhijack's development. The end goal is to be able to anonymously inject shared objects. The libhijack project is maintained by the SoldierX community.

## Previous Research

All prior work injects code into the stack, the heap, or existing executable code. All three methods create issues on today's systems. On AMD64 and ARM64, the two architectures libhijack cares about, the stack is non-executable by default. The heap implementation on FreeBSD, `jemalloc` creates non-executable mappings. Obviously overwriting existing executable code destroys a part of the executable image.

PLT/GOT redirection attacks have proven extremely useful, so much so that read-only relocations (RELRO) is a standard mitigation on hardened systems. Thankfully for us as attackers, FreeBSD doesn't use RELRO, and even if FreeBSD did, using `ptrace` to do devious things negates RELRO as `ptrace` gives us God-like capabilities. We will see the strength of PaX NOEXEC in HardenedBSD, preventing PLT/GOT redirections and executable code injections.

## The Role of ELF

FreeBSD provides a nifty API for inspecting the entire virtual memory space of an application. The results returned from the API tells us the protection flags of each mapping (readable, writable, executable.) If FreeBSD provides such a rich API, why would we need to parse the ELF headers?

We want to ensure that we find the address of the system call instruction in a valid memory location.[22] On ARM64, we also need to keep the alignment to eight bytes. If the execution is redirected to an improperly aligned instruction, the CPU will abort the application with SIGBUS or SIGKILL. Intel-based architectures do not care about instruction alignment, of course.

PLT/GOT hijacking requires parsing ELF headers. One would not be able to find the PLT/GOT without iterating through the Process Headers to find the Dynamic Headers, eventually ending up with the `DT_PLTGOT` entry.

We make heavy use of the `Struct_Obj_Entry` structure, which is the second PLT/GOT entry. Indeed, in a future version of libhijack, we will likely handcraft our own `Struct_Obj_Entry` object and insert that into the real RTLD in order to allow the shared object to resolve symbols via normal methods.

Thus, invoking ELF early on through the process works to our advantage. With FreeBSD's `libprocstat` API, we don't have a need for parsing ELF headers until we get to the PLT/GOT stage, but doing so early makes it easier for the attacker using libhijack, which does all the heavy lifting.

[19] Procedure Linkage Table/Global Offset Table

[20] `unzip pocorgtfo17.pdf phrack56-7.txt`

[21] `unzip pocorgtfo17.pdf phrack59-8.txt`

[22] `syscall` on AMD64, `svc 0` on ARM64.