Attacking the PIC16F1936 code protection

Federico Cerutti, fce201
federico@ceres-c.it

July 8, 2023

Abstract

PIC microcontrollers are widely used in embedded systems. They support a basic code protection mechanism that can be used to prevent unauthorized access to the program memory. This report describes my (unsuccessful) attempt to bypass this protection on a PIC16F1936 microcontroller via voltage glitching.

1 PIC features

PIC microcontrollers are a family of microcontrollers produced by Microchip Technology Inc. All the devices, except for the modern PIC32 series, are Harvard architecture devices, and have variable word sizes depending on the generation. The PIC16 series, the focus of this report, uses 14-bit program words and 8-bit data words.

Differently from other microcontrollers, PICs do not have a bootloader in ROM or memory, and the firmware is executed directly from the program memory. This means that the boot process, as well as debug features are implemented in hardware.

1.1 Memory protection

Most PICs support separated data and program memory protection mechanisms, and the implementation varies depending on the device. The PIC16F1936, has a protection bit for each memory type in Configuration Word 1, a specific 14-bit program memory page in the Configuration Memory space. This page also contains other configuration bits that are required to configure the microcontroller at boot time, such as if the chip should output its internal clock on a pin, or if the brownout reset should be enabled.

When the Code Protection bit is set to 0, the whole program memory is protected, and any read to the program memory with an external programmer will return `0x00`s. The protection feature covers the entire memory of the device, with no per-page or per-block granularity options.

The only way to reset the Code Protection bit to the default value of 1 is to issue a bulk erase program memory command, that will erase the entire program memory. Of course, the code protection does not prevent the code running on the device from reading the program memory.

1.2 ICSP

This is Microchip's proprietary protocol to program PIC microcontrollers, detailed in [1] specifically for the PIC16F1936. The electrical interface is similar to I2C, with two wires (data and clock) and a shared data channel. The control protocol is however different: it does not support any kind of bus arbitration or addressing, there is no concept of read/write commands, and the microcontroller is always the slave.

The boot sequence in programming mode, whose power trace is shown in Figure 1, is as follows:

1. Power on the Device Under Test (DUT)
2. Pull and hold the $\overline{\text{MCLR}}$ pin to V_{IL}
3. Send the 4-bytes key "`MCHP`" to the device
4. Execute the programming commands

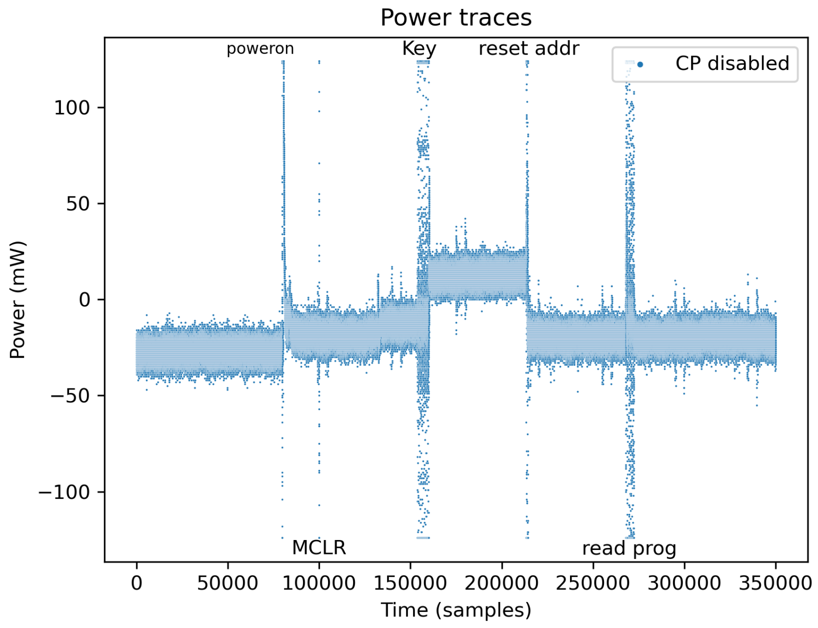


Figure 1: Power trace of a programming session

1.3 Previous work

No previous work on this specific device or chip series has been found, but different PICs were targeted in the past.

Other chips of the PIC family implemented code protection differently, for example allowing to set code protection on memory blocks (groups of pages). As it was possible to erase a single block, it was also possible to clear the code protection on that block, and then to program it with new code. This is an attack vector, as it allows to fit a small program in the newly unprotected block that can then dump over the serial port the content of the remaining protected blocks [2]. This approach is not feasible on the PIC16F1936, as the code protection is not implemented on a per-page basis, and write/erase commands at page granularity are ignored when the global code protection is enabled.

Others managed to bypass the code protection shining UV light on the memory cells that store the code protection bit [3], thus erasing them ionizing the silicon in the transistor's gate. I have decided not to follow this approach as it is necessary to deal with strong acids to decap the chip. While not impossible per se, I wanted to avoid it if possible.

Someone already attempted, apparently unsuccessfully, to bypass the code protection by means of voltage glitching [4]. The presentation's slides are not available, and I have not been able to contact the author to ask for more information.

2 Hardware

I have decided to build my glitching setup around the MAX4619 [5] high speed switch. This chip can be used to switch the power supply of the target between two different sources, and should be fast enough to be used for voltage glitching. It was successfully used by others to attack more modern, capable microcontrollers [6].

The switch is controlled by a Raspberry Pi Pico, which was picked due to its availability and low cost. Also, the PIO subsystem of the Pico allows to generate very precise pulses, which is a requirement for any kind of glitching. This will be discussed in section 3.

The schematic for the final version of the hardware can be seen in Appendix A.

3 Software

To keep things simple and minimizing unnecessary complexity, the software component project is made of two main parts that talk over USB serial.

3.1 Pico firmware

The firmware running on the Raspberry Pi Pico is written in plain C on top of the `pico-sdk`[1]. This program handles the low level details of both glitching and communicating with the DUT.

The Pico is an ARM Cortex M0, and as such supports eXecute In Place (XIP), with an additional cache to allow executing at the nominal 125 MHz clock speed. The cache introduced timing variability, thus it was necessary to disable it to obtain consistent results; some key functions were moved in RAM.

3.1.1 PIO

The IC on the Raspberry Pi Pico board, the RP2040, supports a novel feature called PIO, or Programmable IO. There are 2 PIO blocks, with 4 state machines each, that can be programmed to perform simple tasks, such as generating a pulse of a specific length, or reading a value from a pin and storing it in a register. These state machines run independently of the CPU, and each instruction takes a single clock cycle to execute.

This feature was designed to add support for a variety of peripherals/protocols while maintaining the chip silicon area low. Essentially, it allows to forgo the omnipresent bitbanging of GPIOs to implement custom protocols.

3.1.2 Glitching

Glitching requires accurate timing, and the PIO state machines are perfect for this task. A PIO block is used to execute the following algorithm:

1. Wait for a specific waveform (rising or falling edge) on the trigger input pin
2. Count a given number of clock cycles: the glitch delay
3. Pull the MAX4619 control pin low: DUT V_{DD}= glitch tension.
   Optional: Pull up the trigger output pin high
4. Count a given number of clock cycles: the glitch width
5. Pull the MAX4619 control pin high: DUT V_{DD}= nominal tension.
   Optional: Pull down the trigger output pin

Once the state machine is configured with the necessary parameters (trigger waveform, delay, width), execution returns to the main program which can continue with its tasks, such as enabling programming mode and issuing commands to the target: the glitch will execute asynchronously.

3.1.3 ICSP

To break the code protection feature of the chip, it is necessary to issue commands to the target in programming mode. While ICSP is a proprietary protocol, many open source implementations exist, but most of them bitbang the GPIOs, taking a lot of CPU time and introducing timing variability.

I have then decided to implement ICSP in PIO assembly code, to minimize the timing variability and to allow the CPU to perform other tasks while the ICSP commands are executed. ICSP functionality is exposed on the same USB serial interface, and the glitch firmware can also be used as a standalone PIC programmer.

3.2 Controller software

The controller software is written in Python, and is responsible for orchestrating the glitching process. It accepts options from the command line, and then configures the Pico firmware accordingly communicating over USB serial.

[1] `https://github.com/raspberrypi/pico-sdk`

To optimize the glitching process, the search space is explored in random order, and the results are plotted in real time with a QT scatter plot. This allows to monitor the ongoing glitch campaign, and to optimize the parameters as appropriate.

3.3 ICSP python module

A small python module is available in `src/icsp`. It buils upon the firmware feature described in 3.1.3, and offers a simple interface to communicate with PIC microcontrollers over ICSP. In the same folder there is also a device library which offers an organized structure, and allows to support other PICs.

3.4 Trace acquisition

To capture traces from my oscilloscope, I used pyVISA[2], a python Virtual Instrument Software Architecture (VISA) implementation that allows to control test equipment over USB, Ethernet, GPIB, etc. Once I identified the correct trigger settings on the oscilloscope, the device could be remotely configured to perform acquisitions in a loop. Each trace is then retrieved and stored in a python array in a Jupyter notebook, since I was acquiring a limited amount of data (200 traces) I had enough memory not to save the traces to disk.

The same Jupyter notebook was also used to perform the analysis of the traces that will be discussed later using numpy, matplotlib and seaborn.

4 Glitching approach

4.1 Arithmetic loop

To validate my setup and verify if the microcontroller could be glitched via voltage glitching, I programmed the target with a simple firmware that would run a loop with arithmetic operations.

```
IO_RA0_SetHigh();           // Keep output pin high during for loop execution
for(a = 0; a < 0x10; a++) b--;
if (a == 0x10 && b == 0) {  // First set either output pin
        IO_RA1_SetHigh();
} else {
        IO_RA2_SetHigh();
}
IO_RA0_SetLow();           // Then finally pull the loop pin down
```

Listing 1: Firmware A

This program allowed the glitcher to sync with the DUT sensing **PORTA** pin 0, and to easily detect if the glitch was successful or not with **PORTA** pins 1 and 2.

The above code was indeed glitchable with the following parameters:

- **Power supply**: 3.3V
- **Glitch voltage**: 2.25V
- **Glitch width**: 20 Pico cycles

The above parameters were found essentially by trial and error, trying to glitch the target with a wide range of widths and delays, and then adjusting the voltage on the lab power supply. Once I had around 20% of brownouts, I slightly reduced the glitch width, and then I was able to glitch the DUT consistently. Judging by the oscilloscope traces, most of the glitches resulted in the loops being shorter, which means that most likely the glitch was targeting the ALU and resulting in wrong comparisons.

[2] https://github.com/pyvisa/pyvisa

4.2 Flash readout

Encouraged by the success of the previous experiment, I tried to glitch program memory (flash) reads. The reasoning behind this is that the code protection bit is stored in a Configuration Word in a special area of the program memory. Since, as mentioned in subsection 1.1, the Configuration Word contains value relevant to the chip features during early boot, I assumed that it would be read from flash memory at startup, and possibly stored in registers or flip-flops. Glitching the read operation would then result in the chip booting with a different configuration to the one stored in flash, possibly with code protection disabled.

I then modified the software in the PIC to read out a word from the flash memory, and send it to the glitcher over I2C. In turn, the Pico would forward it to the host software, where it is compared with the expected value. Code in Listing 2 populates two global variables with the 14-bit word from memory which will later be sent to the glitcher by the I2C interrupt handler.

The two while loops at the beginning are used to wait for a positive pulse, which is necessary to avoid a potential race condition when the brownout reset of the target is triggered. If the PIC was not waiting for a pulse, on reset it would immediately fetch a value from memory, this time without the glitch, and send it over I2C to the glitcher, which would be still waiting for the value from the previous boot, resulting in a false negative.

```c
#define FLASH_READ_ADDR 0xAAA

while(!IO_RA1_GetValue()) asm("nop");   // Wait for toggle high...
while(IO_RA1_GetValue())  asm("nop");   // ...and low

I2C_Open(); // From now on we can return data over i2c
I2C_SlaveSetWriteIntHandler(ReadIntHandler);
I2C_SlaveSetAddrIntHandler(AddressIntHandler);

// Configure address to read from
EEADRL = (FLASH_READ_ADDR & 0x00FF);
EEADRH = ((FLASH_READ_ADDR & 0xFF00) >> 8);

EECON1bits.CFGS = 0;    // Deselect Configuration space
EECON1bits.EEPGD = 1;   // Select Program Memory

IO_RA0_SetHigh();       // Keep output pin high during memory read operation
EECON1bits.RD = 1;      // Initiate Read
NOP();
NOP();
IO_RA0_SetLow();        // Then pull the loop pin down
datl = EEDATL;
dath = EEDATH;

while(1) asm("nop");
```

Listing 2: Firmware B

I optimistically expected to be able to glitch the target as easily as before, potentially with minor adjustments to the glitch parameters, but unfortunately this was not the case. I did manage to get wrong data from the program memory, but all the glitches did yield the same unexpected value: `0x0`, the default initialization value for the global variables.

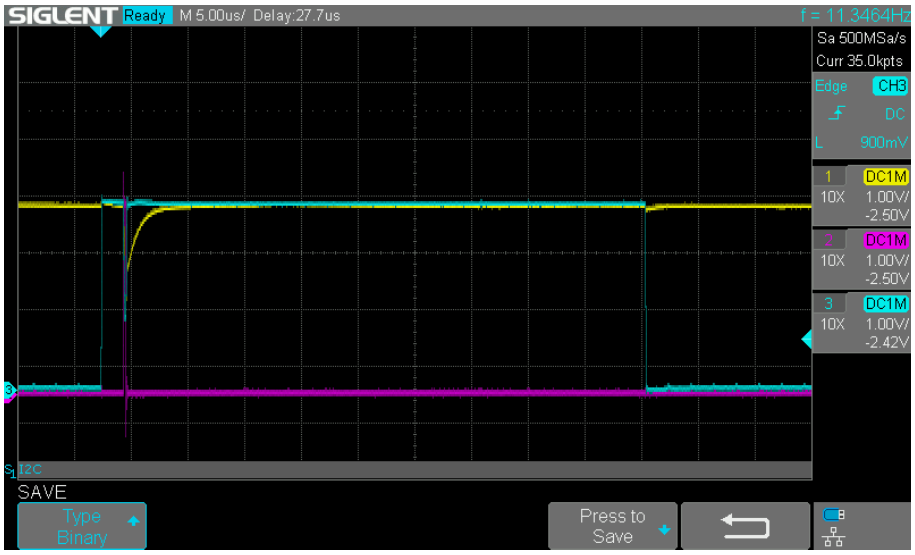


Figure 2: Glitching the flash readout
Channel 1: SCL Channel 2: Glitch trigger out Channel 3: PIC `PORTA` pin 0

As shown in Figure 2, for the glitch to succeed the delay had to be relatively small, thus the glitch had to be triggered very early in the glitch window. Looking at Listing 2, we notice that the output pin is pulled up just before `EECON` register's bit `RD` is set to 1. I speculate that this glitch might be interfering with the register write, effectively not triggering the flash read from the memory subsystem, while the flash peripheral itself operates unaffected.

4.3 Code protection

Not succeeding in glitching the flash readout, I decided to analyze the power consumption of the PIC16F1936 during the boot process with and without code protection enabled.

subsection 3.4 describes the host software setup used to acquire the traces. A specific command was added to the glitcher firmware to guarantee constant execution times to a resolution of 5ns, so that the traces could be aligned in post-processing.

Seeing a difference in the power consumption after a certain amount of time from boot would imply that the code protection bit has been read by that point, and that I could reduce the search space for a suitable glitch. Given that the code protection bit is stored near other values important for the chip configuration, as mentioned in subsection 1.1, I expected to see a difference in power consumption at early boot, potentially even before the $\overline{\text{MCLR}}$ pin is brought to V_{IL}.

The power consumption was measured with an Ohmite APC0805T49R9Z 49,9Ω precision resistor in series with the ground pin of the PIC, and my Siglent SDS1104X-E oscilloscope. Having this oscilloscope an 8-bit ADC, the measurements are probably not very accurate. Furthermore, the ground pin of the PIC is shared with the ground pin of the Raspberry Pi Pico, thus with my laptop. This setup can't by any means be considered accurate, however it should be good enough to see a difference in power consumption between the two configurations with.

To compare the traces, I borrowed the mathematical tools from Correlation Power Analysis (CPA), namely the Pearson correlation coefficient (Equation 1) and the Hamming weight of the difference between the two traces. I chose the Hamming weight over Hamming distance because I expected the register or flip-flop to start in an empty state, thus the two leakage models would be equivalent.

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{1}$$

Acquiring 100 traces per configuration, I stored them in the Measurements matrix MM, where each row is a trace, and each column is a sample. I also have the Power-prediction matrix PP, a $(200, 1)$

column matrix with the hamming weights (HW) of the Configuration Word 1 with and without code protection.

$$MM = \begin{bmatrix} t_{\overline{CP}1}[1] & t_{\overline{CP}1}[2] & \dots & t_{\overline{CP}1}[350000] \\ \vdots & \vdots & \ddots & \vdots \\ t_{\overline{CP}100}[1] & t_{\overline{CP}100}[2] & \dots & t_{\overline{CP}100}[350000] \\ t_{CP1}[1] & t_{CP1}[2] & \dots & t_{CP1}[350000] \\ \vdots & \vdots & \ddots & \vdots \\ t_{CP100}[1] & t_{CP100}[2] & \dots & t_{CP100}[350000] \end{bmatrix} \qquad PP = \begin{bmatrix} \mathrm{HW}(\texttt{0x3FFF}) \\ \vdots \\ \mathrm{HW}(\texttt{0x3FFF}) \\ \mathrm{HW}(\texttt{0x3F7F}) \\ \vdots \\ \mathrm{HW}(\texttt{0x3F7F}) \end{bmatrix}$$

The two matrices are correlated by column with Equation 1, and the results are plotted in Figure 3 along with a single powertrace to easily identify the boot stages. When the correlation value is close to 1, power consumption in that instant is directly related to the state of the code protection bit. Comparing this picture with Figure 1, we notice that there is high correlation in 3 key moments:

- The first peak, when the chip is powered on. At this resolution, this is probably due to measurement error, but further analysis could be done focusing on this moment.
- After the ICSP key has been sent to the target, and before first ICSP command is issued. I have no explanation for this, but it signals the fact that by this point the chip already knows if code protection is enabled or not.
- The last peak is the data readout from program memory. This is expected, as no data will be sent to the programmer if code protection is enabled, thus power consumption will be lower.

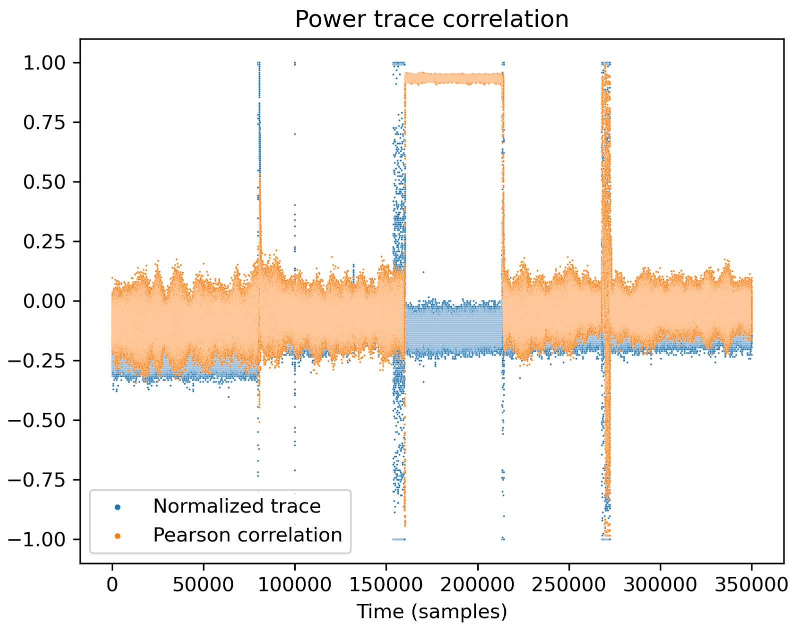


Figure 3: Pearson correlation coefficient between the power traces with and without code protection

The second peak is the most interesting, as it tells us that the check for code protection status is probably done once at boot, and not on a per-read basis. This is promising as, if a glitch is successfully obtained, it will be possible to read the entire memory. Otherwise, it would have been necessary to repeat the glitch for each read.

What is missing in this picture is a clear correlation during early boot, something similar to an if-then branch being executed right after the code protection bit is read, which I had hoped to find to pinpoint the exact moment when the code protection bit is read. This was somewhat expected since, as stated in section 1, these devices do not employ a bootloader.

5 Lessons learned and future work

The order of steps in section 4 do not *strictly* reflect the chronological order in which events unfolded during this project. In fact, I tried to attack the code protection right after I managed to glitch the arithmetic operations. After some days of wondering why I was getting no results, I noticed my fundamental mistake: I had characterized the wrong operation. The results of subsection 4.2 explain why I was unsuccessful in glitching the code protection bit. The **first** lesson learned is: carefully evaluate if your model is correct.

Then, I have to say I learned a lot about how to build all of this myself. I had never brought up a measurement

Other glitching techniques must be considered in the future: I only tried dowvolting the device, but it might be possible to overvolt it for short period of times during the flash readout, for how dangerous this might be to the chip. Clock glitching is not suitable as the chip runs on its internal oscillator during flash readout operations. Electromagnetic fault injection might be suitable, as there are multiple peripherals that could be affected: flash, memory bus, or even the internal clock. For this I am planning to build a ChipSHOUTER-PicoEMP board[3] in the near future.

Finally, in Figure 4 there is a capture of the PIC16F1936 die, taken with my cheap electronics repair HDMI microscope. The chip was decapped[4] at the hackerspace in Den Haag[5]. I want to obtain actually useful pictures in the future with either a metallurgic or scanning electron microscope.

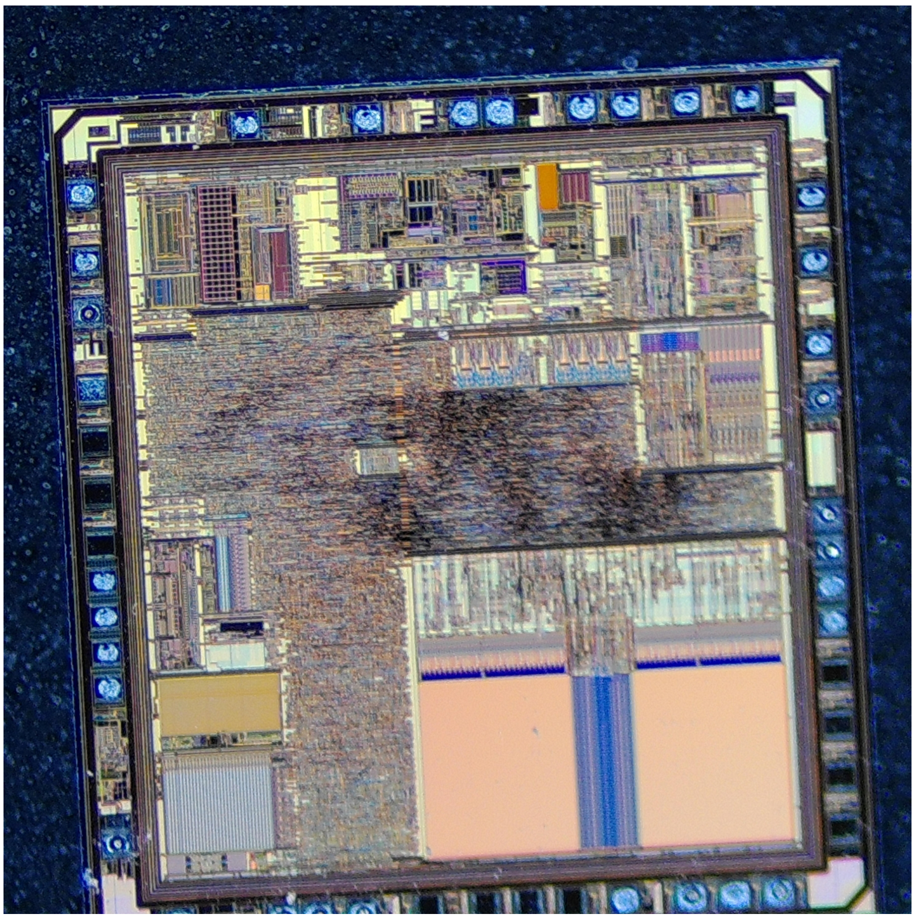


Figure 4: PIC16F1936 silicon die

[3]`https://github.com/newaetech/chipshouter-picoemp`
[4]Thanks a lot to Peter Bosch `@peterbjornx`
[5]`https://revspace.nl/`

Acronyms

CPA Correlation Power Analysis. 6

DUT Device Under Test. 2–4

XIP eXecute In Place. 2

Glossary

Hamming distance The number of bits that are different between two words. 6

Hamming weight The number of bits being non-zero in a word. 6

V_{IL} Voltage Input Low - The maximum voltage that is considered a logic low. 1, 5

References

[1] *PIC16F193X/LF193X/PIC16F194X/LF194X/PIC16LF190X Memory Programming Specification*, https://ww1.microchip.com/downloads/en/DeviceDoc/41397B.pdf, Microchip Technology Inc., rev. B.

[2] M. Meriac, "Heart of darkness-exploring the uncharted backwaters of hid iclass (tm) security," in *24th Chaos Communication Congress*, 2010.

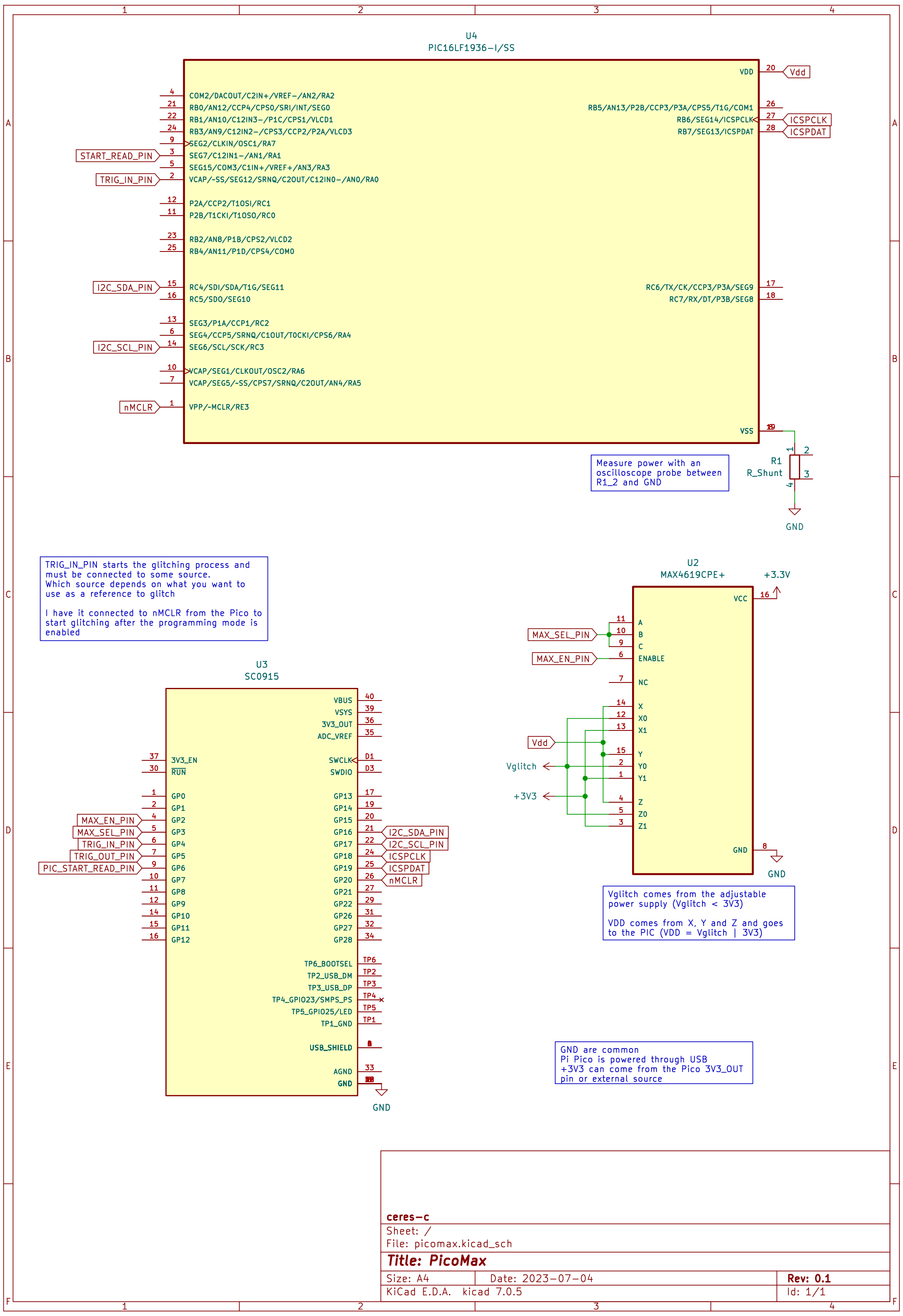
[3] A. "bunnie" Huang, "Hacking the PIC 18F1320," https://www.bunniestudios.com/blog/?page_id=40, 2007.

[4] S. Cesare, "Adventures in glitching PIC microcontrollers to defeat firmware copy protection," in *Kiwicon*, 2015.

[5] *High-Speed, Low-Voltage, CMOS Analog Switch*, https://datasheets.stg-maximintegrated.com/en/ds/MAX4617-MAX4619.pdf, Maxim Integrated Products, Inc., rev. 3.

[6] C. Gerlinsky, "Breaking code read protection on the nxp lpc-family microcontrollers," 2017, rEcon.

A Hardware



Measure power with an oscilloscope probe between R1_2 and GND

TRIG_IN_PIN starts the glitching process and must be connected to some source. Which source depends on what you want to use as a reference to glitch

I have it connected to nMCLR from the Pico to start glitching after the programming mode is enabled

Vglitch comes from the adjustable power supply (Vglitch < 3V3)

VDD comes from X, Y and Z and goes to the PIC (VDD = Vglitch | 3V3)

GND are common
Pi Pico is powered through USB
+3V3 can come from the Pico 3V3_OUT pin or external source

ceres-c
Sheet: /
File: picomax.kicad_sch
Title: PicoMax
Size: A4 Date: 2023-07-04 **Rev: 0.1**
KiCad E.D.A. kicad 7.0.5 Id: 1/1