# ESP32 Remote Reset & Programming Fix

## Background

Most ESP32 development systems, including ESP32-DevKitC-v4, utilize a two-transistor circuit to allow for remote reset and bootloader triggering for code download via the USB port. The objective is to use digital signals of the SiliconLabs CP2012 USB-Serial bridge to control this behavior. Conveniently, two digital signals are available and otherwise unused on DevKits: DTR & RTS. These signals are intended to be used for “hardware flow-control” during serial communication. The DevKits do not implement hardware flow-control, so are available for other uses and are easily controlled by the Host operating system. Because these signals are part of historical serial communication hardware and software implementations, they are convenient for the above task and almost universally controllable across all Host operating systems.

The circuit was designed to solve two issues of convenience for these DevKits: 1. Remote Reset and trigger of Bootloader or Application code. 2. Mitigating the problem of DTR & RTS potentially both being asserted (or deasserted) by default behavior of serial port initialization for some (many?) Host OS driver implementations.

Unfortunately, it’s not a straight path from having these signals available and using them as desired. As DTR & RTS are driven both HI and LO by the CP2012, they can not be tied directly to the **EN** and **IO_0** pins without sacrificing functionality of the device. **EN** is considered an “Open Drain” signal which means many devices might assert this signal (LO), including the ESP32 itself. **IO_0** is a generally used GPIO signal, and would be a wasted pin if it was dedicated to only being used for determining boot mode. Hence the circuit of **Figure 1**.

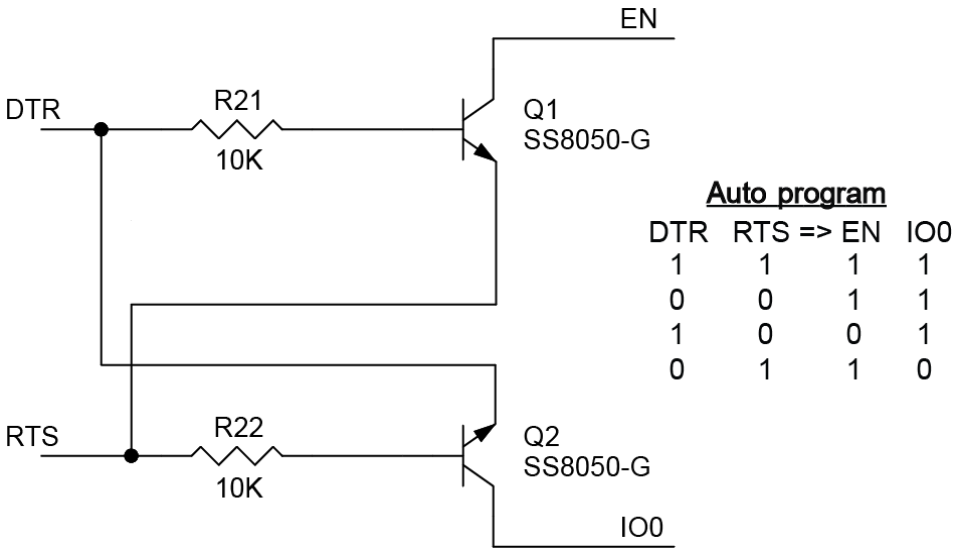



**Figure 1: Typical ESP32 DevKit Reset/Programming Circuit**

ESP32 DevKits commonly implement the circuit of **Figure 1** to control the **EN** and **IO_0** signals. Notice from the truth table that there is no condition where any combination of input signals can cause both **EN** & **IO_0** to be asserted (LO) simultaneously. It is possible to cause *EITHER* **EN** or **IO_0** to be asserted LO**,** but *NOT BOTH***.** Another nuance of the truth table is that 0=LO (driven), 1=Hi-Z (not driven) by the circuit. When observed with an oscilloscope, the “1” state will be seen as a HI because of pull-up resistors on those signals outside of this circuit, however the circuit of **Figure 1** will not be influencing the signals. Configuring the circuit in this manner allows **EN** to be used as intended, and except for boot, **IO_0** may be used by the application.

After a system reset, the ESP32 samples the **IO_0** pin microseconds after **EN** is deasserted to determine whether the ESP32 will execute application code (**IO_0** = HI) or Bootloader (**IO_0** = LO). After **IO_0** is sampled it is no longer regarded during the boot process. There is a narrow window during which this signal is sampled, and this is where the nuances of device driver and Host operating system implementations interact with the circuit itself.

Despite universal use of SiliconLabs CP2012 USB-Serial bridge devices on the DevKit boards, each Host operating system has its own unique device driver implementation. Among the differences in implementation are varying latencies between system-call and state-change of these signals (ie. the time it takes from function call to actually causing the signals to change). For typical serial communication this poses no problem because those time scales are small compared to serial data rates. However, when we repurpose these signals for another use, these differences can impact desired behavior. This means on some OSs the signals can change state almost instantly, while in other cases there is some amount of delay/latency from command to state change. This latency can cause “space” between changing of one signal and the other and is what causes the bootloader triggering not to function as intended.

One typical workaround is to press/hold a button on the ESP32 DevKit to hold **IO_0** LO while executing the firmware download command sequence. This works because it guarantees that **IO_0** will be sampled as LO during the process and that Bootloader will be entered to enable download. Because **IO_0** is otherwise ignored, this works. However, if the device is mounted inside an enclosure, or as part of a CI/CD or automated test workflow, this removes the ability to automatically download firmware.

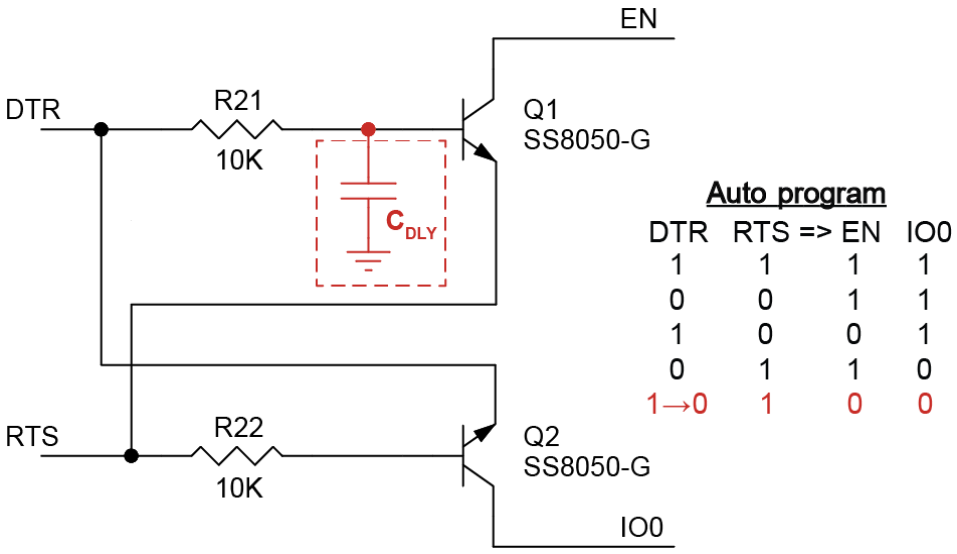
Another popular mitigation is the addition of a large value capacitor to the **EN** (reset) signal on the DevKit board. This slows the transition from LO=>HI of the **EN** signal which has a side-effect of “delaying” when the ESP32 sees **EN** become HI. The working idea here is that this delay can cause the **IO_0** signal to be sampled at a later time, presumably after any latency/delay caused by the Host operating system. This mitigation has been somewhat successful to solve the automatic firmware download problem, but causes other potentially unwanted system level issues.

The problem with this strategy is that it stretches the rise time of the **EN** signal to be VERY long. This may have detrimental effects on system stability. When the ESP32 is integrated into a larger system, and multiple devices rely on the **EN** signal to reset to their initial operating state, devices may come out of their reset state in unpredictable order due to differing thresholds on their logic inputs. Further, many devices may enter a “metastable” or indeterminate state while **EN** transitions, leaving the entire device in an unknown operating state. Since each device in a system using this signal may behave differently to a slowly rising **EN** signal, it may cause inconsistent behavior each time the system is initialized. So, while this mitigation is also a poor solution, it hints at a potential fix for the on-board circuit.

# The Fix

**Figure 2** below shows the typical circuit found on the DevKit board, with the addition of a capacitor called $C_{DLY}$. This capacitor is charged during the first part of the programming cycle (**DTR** = HI) and supplies current to Q1 during the timing gap between **DTR** 1⇒0 and **RTS** 0⇒1 which can vary from platform to platform. The addition of this capacitor creates a transient "5th state" in our truth table, which is shown in Figure 2 in red. Further, while not noted in the truth table, we also must account for another behavior of the ESP32 to fully understand how this circuit behaves. We will dive into these details in the "How it Works" section.




Auto program

| DTR | RTS => EN | | IO0 |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1→0 | 1 | 0 | 0 |

**Figure 2: Modified ESP32 DevKit Reset/Programming Circuit**

Through the addition of $C_{DLY}$ we effectively extend the amount of time that **EN** is asserted LO by keeping transistor Q1 turned on even after **DTR** is set to 1. The value of $C_{DLY}$ is selected to "bridge the gap" between the setting of the **DTR** & **RTS** signals due to Host OS latency. Once **RTS** = 1, Q2 is immediately turned on, Q1 immediately turned off, forcing **IO_0** = LO, and **EN** = HI, essentially simultaneously. This occurs because the Emitter of Q1 is connected to **RTS**, which means the single transition of **RTS** controls the state change of both **IO_0** and **EN**, hence causing them to occur simultaneously. (Q1 is no longer controlled by the charge on $C_{DLY}$) Furthermore, the transition time of **EN** is largely controlled now by the transition time of **RTS**, so it happens at logic-level transition speeds.

In our experience, latency between transitions of DTR & RTS can be anywhere from 1ms to 20ms depending on the OS and software stack utilized. We have simulated these delays using LTSpice and tested on actual systems and come to the conclusion that an actual capacitance of ~22uF is sufficient for most cases. However, because of variation in actual values of typical MLC Capacitors, we suggest a capacitor value of 33uF. This assures that Q1 is kept on for the duration of time required.

In cases where simply a "RESET" is required (only affecting **EN**) the capacitor $C_{DLY}$ simply adds a small amount of latency, typically a few ms from the time **DTR** is deasserted until the system comes out of Reset. This is typically similar to what happens when using large-value capacitance directly on the EN signal line and has no detrimental effect on typical operation.

# Implementing the Fix

Since all ESP32-DevKits as of the time of this writing implement the typical circuit, we can apply this fix with a small PCB-hack on the ESP32-DevKit itself by adding a MLC Capacitor to the right circuit node. We have only tested this on ESP32-DevKitC-v4, but others should be similar. The photos here are for ESP32-DevKitC-v4.

In order to implement this fix in the field, one must:

1. Locate Q1 on the PCB
2. Identify the Base terminal of Q1
3. Find a suitable connection to GND
4. Solder $C_{DLY}$ between the Base pin of Q1 and GND

**ESP32 DevKitC-v4 Remote Reset Rework**

In our implementation of these fixes, we use an 0805-sized MLCC for $C_{DLY}$. Any size capacitor which fits in the space may be used.

- Remove solder mask as indicated by the copper area shown in the figures.
  The **back** of an Xacto blade is good for this.
- Solder 33uF 0805-size MLCC capacitor to the base of transistor Q1 as shown.
- Solder other end of cap to PCB ground plane on cleared copper area

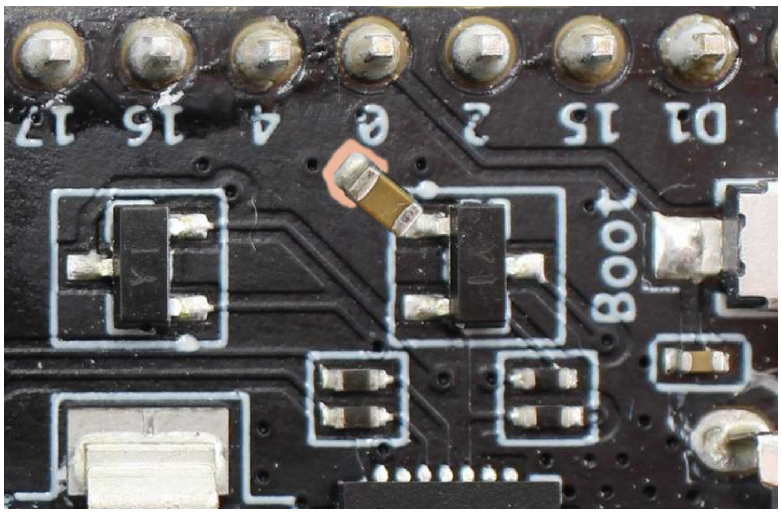



**Figure 3: Completed Rework on ESP32-DevKitC-v4**

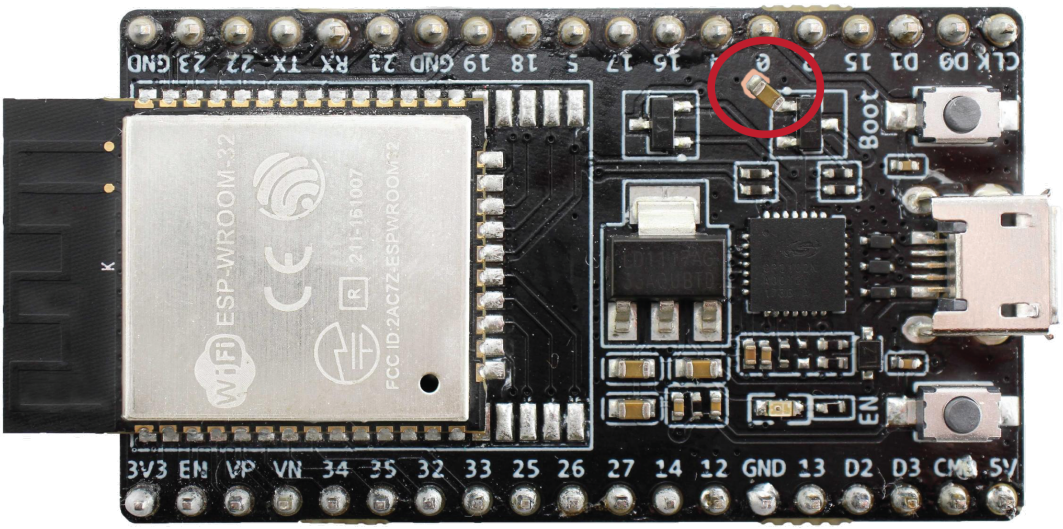



**Figure 4: Overview of Reworked ESP32-DevKitC-v4**

# How it Works:  Theory of Operation

For those who are interested in the circuit level details and timing simulations of the circuit, we present this section with deeper circuit description and timing diagrams derived from the simulation. We also provide the LTSpice program for those who would like to experiment on their own.

We start by presenting a series of timing diagrams illustrating both the ESP32 device and the reset circuit operation, the problem, and the various mitigations. All of these diagrams were extracted from the ESP32 datasheet or generated using LTSpice. Simulations are used primarily for expedience in demonstrating the various cases with sufficient fidelity to illustrate the issues. At the end we will show a scope capture of the fix in action. Please note all of these simulations and characterizations are based upon a 3.3V supply voltage to the ESP32, as is present on the DevKits.

# Reset on the ESP32 and the “Uncertainty-Zone”

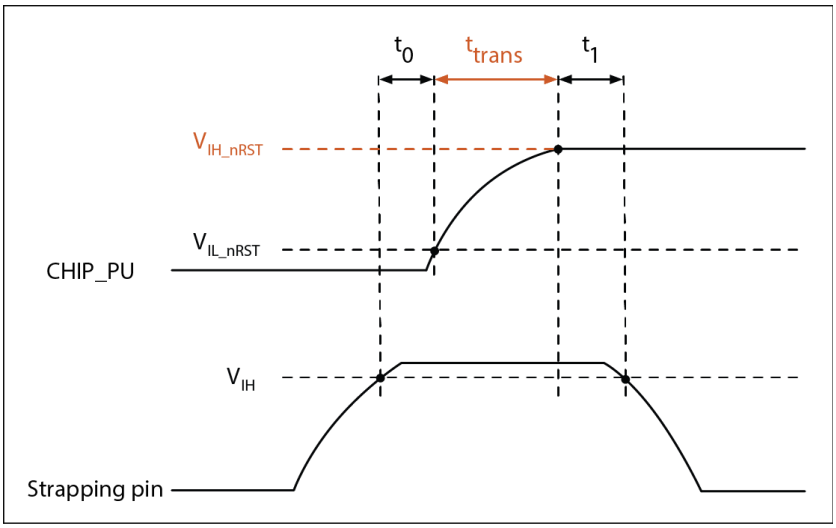



**Figure 5: ESP32 Strapping Pin Setup and Hold Times**

Here we see in **Figure 5,** a timing diagram extracted from the ESP32 device datasheet with a couple of added parameters. Of note, these unspecified parameters have bearing on and interact with our reset circuit and are important to consider. Here in **Table 1** we summarize these parameters, extracted from the ESP32 datasheet. Both **nRST** and **CHIP_PU** are the same signal as **EN** on the DevKit boards.

| Param | Description | Value |
|---|---|---|
| $t_0$ | Setup time before EN transition LO-to-HI | 0 ms |
| $t_1$ | Hold time after EN is HI | 1 ms |
| $t_{trans}$ | Transition time of EN from LO to HI | Unspecified |
| $V_{IL\_nRST}$ | Low-level input voltage of CHIP_PU to power off the chip | 0.6 V |
| $V_{IH\_nRST}$ | Voltage where CPU Core begins startup process | Unspecified |

**Figure 5: ESP32 Strapping Pin Setup and Hold Times**

Examining **Table 1**, we see some important parameters don't have specified values. The parameters $t_{trans}$ and $V_{IH\_nRST}$ play an important role in telling us how the ESP32 CPU starts up. A slow transition from LO to HI on **EN** ($t_{trans}$) means the signal seen by the ESP32 and internal circuitry spends quite a bit of time between the voltage thresholds $V_{IL\_nRST}$ and $V_{IH\_nRST}$. This is significant because of a phenomenon called Metastability (check it out: https://en.wikipedia.org/wiki/Metastability_(electronics) ). Without getting into too many details, even portions of the internal circuitry of the ESP32 could start "operating" at slightly different voltage levels on the **EN** signal, which means the more time spent in this the "uncertainty-zone" between $V_{IL\_nRST}$ and $V_{IH\_nRST}$, the higher the chances that "weird things can happen". These uncertainties are multiplied when one considers other devices outside of the ESP32 which use the **EN** signal for reset/initialization. As a result, it is important to minimize the time $t_{trans}$ by forcing **EN** to transition from LO to HI as quickly as possible. All of this is significant to make the argument that attaching a capacitor on the **EN** signal to slow this transition is a risky fix because of the uncertain side effects. Below is a comparison of the transition of the **EN** signal with and without an attached 1uF capacitor.

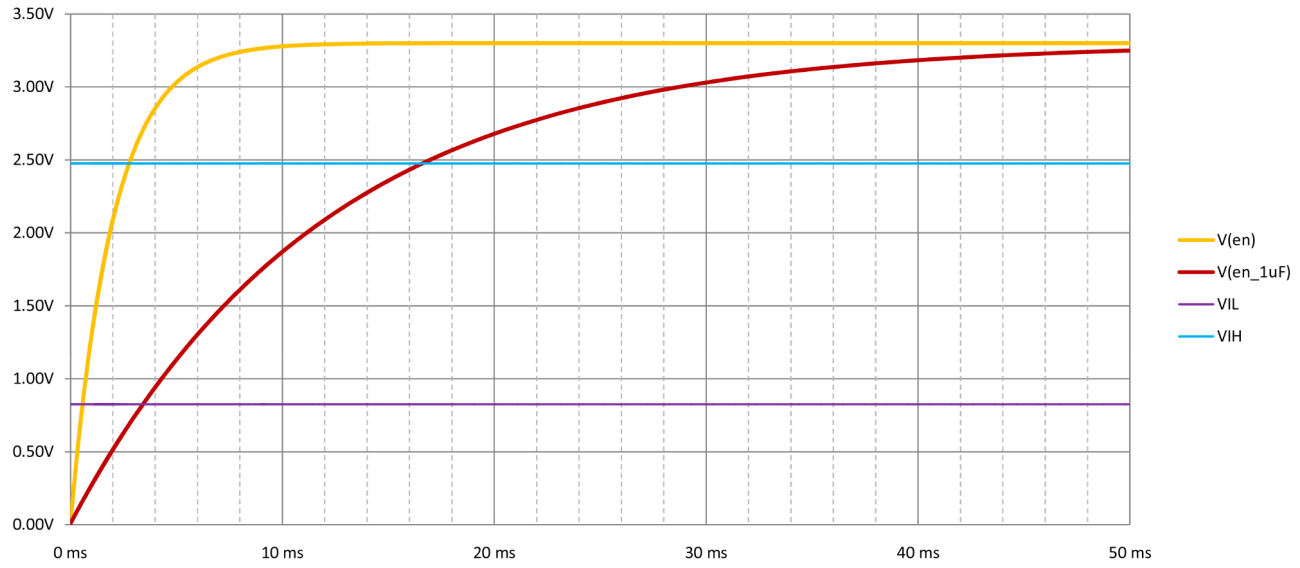


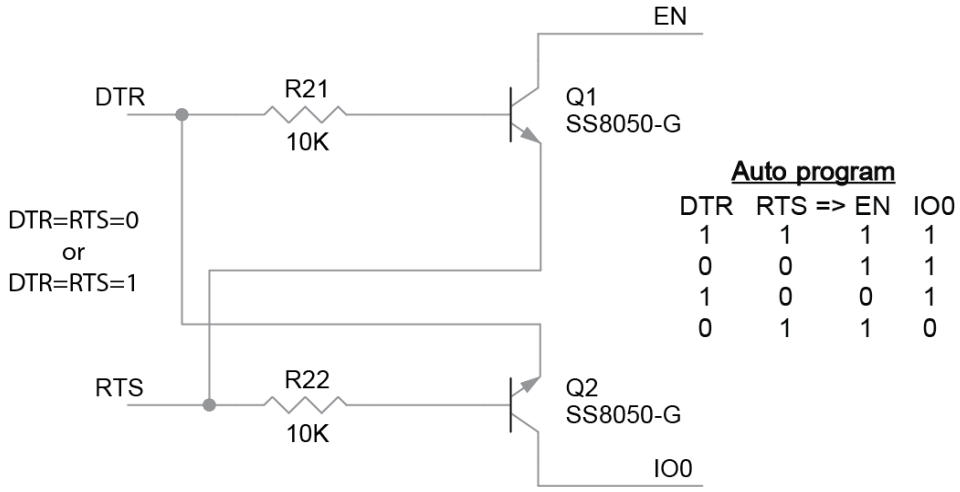

**Figure 6: Impact of Load Capacitor on EN Rise Time**

The space between the Blue and Purple lines represents the "uncertainty-zone" within which we want to spend as little time as possible. The Yellow line represents **EN** rise time typical of the DevKit reset circuit which clocks in at slightly less than 3ms. In contrast, with a 1uF load capacitor, rise time of **EN** is almost 17ms, leading to more uncertainty in system initialization. One might argue "seems to work for me", but this is the problem with uncertainty, it crops up unexpectedly! Our argument is that if there is a way to easily reduce uncertainty in any digital system, it's a no brainer to implement it!

With that discussion behind us, let's continue on now to understand how the current reset circuit works, and how the strategic placement of a capacitor can resolve the problem at hand without pushing us further into the "uncertainty-zone".

# How the Reset Circuit Works

Whomever designed that little 2-transistor reset circuit deserves some credit. It's quite clever and uses a minimal number of components to solve the problems we've mentioned. So, how does it actually do this? We'll try to illustrate with a couple of diagrams.

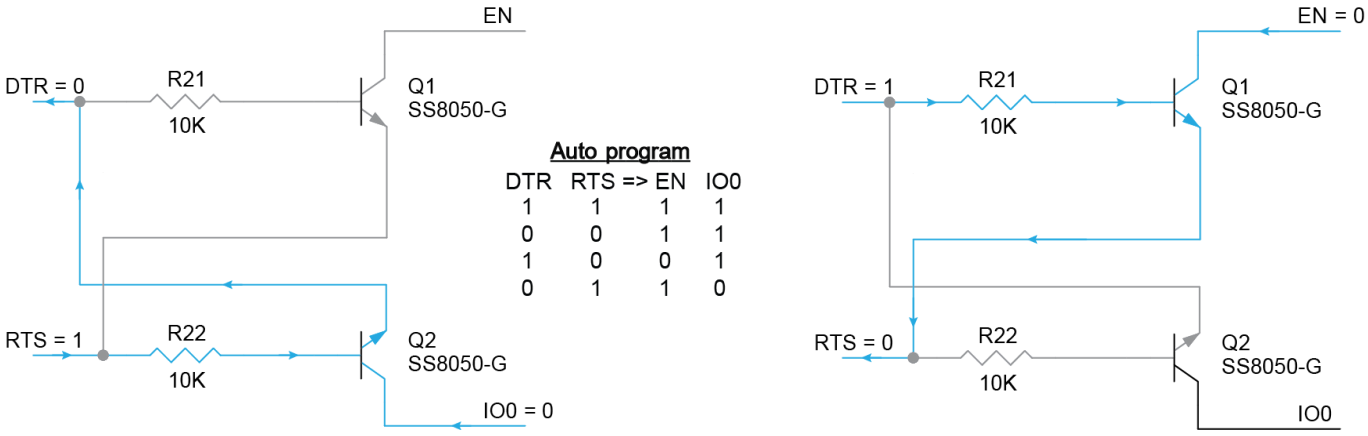



**Auto program**

| DTR | RTS => EN | IO0 |
|---|---|---|
| 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |

**Figure 7: DevKit Reset Circuit when DTR = RTS**

The first case we will consider is when both DRT & RTS are set to the same logic level, the first two lines of the truth table. In both of these cases, because **DTR** & **RTS** to the opposite Base & Emitter of Q1 & Q2, any time both terminals of the transistors are at the same potential they are both off. No current flows in the circuit. From the perspective of **EN** & **IO_0**, the circuit does not exist.




**Auto program**

| DTR | RTS => EN | IO0 |
|---|---|---|
| 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |

**Figure 8: DevKit Reset Circuit when DTR <> RTS**

The next cases we examine are more interesting. The blue arrows in **Figure 8** show how current flows in the circuit when the two signals are different. Here we see that current flows from the signal set to 1 through the BJT to the signal set to 0. This causes the corresponding transistor to turn on, which causes the attached signal to be pulled down to near the potential of the "0" signal. It is through this mechanism that either the **EN** or **IO_0** signal can be asserted LO. The signal not being pulled LO behaves as an "open circuit" and does not affect the associated signal node.

## Looking at the Problem

Now we have analyzed the four “steady state” or stable states of the circuit. The more interesting case emerges when we examine the “transitory state” as we move from line 3 of the truth table to line 4. The original designer of the circuit assumed that the two signals would change logic levels nearly simultaneously. Perhaps on the Host OS and driver stack they tested the circuit with, this was actually the case. However, as the world now knows (and many message board conversations will confirm) this is not always the case. The entire reset sequence happens in these distinct steps:

1. **DTR** = 1, **RTS** = 1, Host OS sends command to set **RTS** = 0
2. Software Delay ~120ms
3. **DTR** = 1, **RTS** = 0, Host OS sends command to set **DTR** = 0
4. **DTR** = 0, **RTS** = 0, Host OS sends command to set **RTS** = 1
5. Software Delay ~150ms
6. **DTR** = 0, **RTS** = 1, Host OS sends command to set **DTR** = 1

As we can see from this sequence, there is a moment when both **DTR** & **RTS** = 0 because only one signal can change at a time (at least in some implementations). We know from our previous analysis that this means that both transistors will be OFF, and signal nodes of **EN** & **IO_0** will behave as if this circuit is not present. So, what happens? This is best explained with a timing diagram.

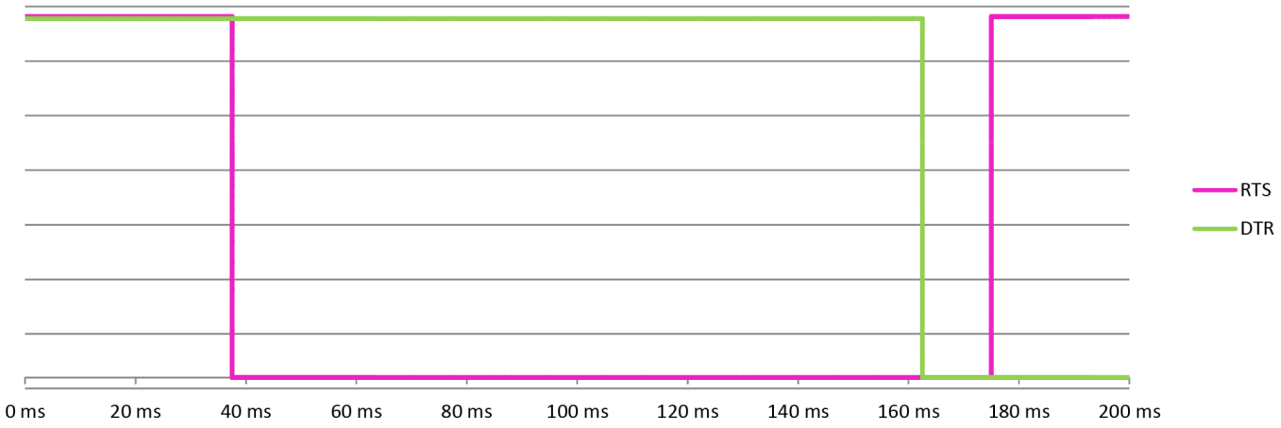



**Figure 9: DTR, RTS Transition Timing**

As shown in **Figure 9**, there can be quite a sizable gap between setting **DTR**=0 and **RTS**=1. In this case about 10ms. This particular case is the result of running the ESP development tools in a VM running on a Windows host. In this case there are two levels of drivers which must be navigated for each request to change to one of our signals. In the Windows-native case this gap can be anywhere from 2ms to 5ms in duration. Reports from people using the tools in Mac-native and Linux-native vary but seem to generally get better results. This means, presumably, the signals can be transitioned with less delay between commands in those Host OS/driver stack combinations. Given the widespread use of Windows as a development platform, and the increasing use of VMs as containers for bodies of work, it seems like a solid solution to this problem will have a positive impact on many users.

The best way to understand how this gap can affect the intended operation of the reset circuit is, of course, with another timing diagram. For purposes of discussion the following diagrams show simulations of the same circuit, but with shorter waiting times (Steps 2 & 5 above), since all of the action happens at the transitions. This allows us to get more detailed pictures of the points of interest.

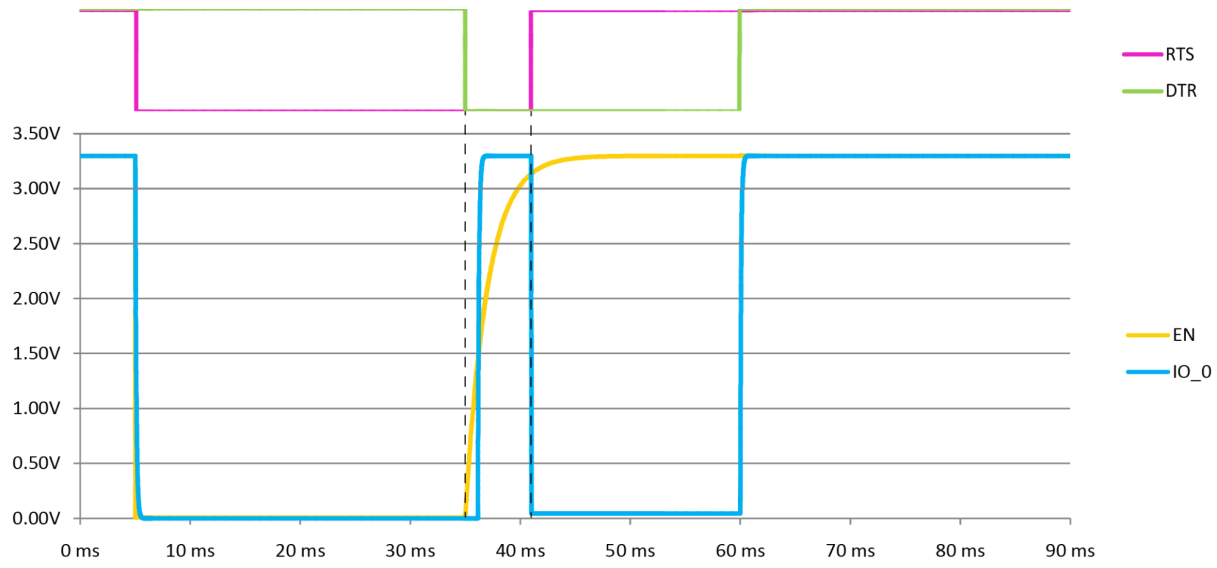



**Figure 10: EN & IO_0 vs. RTS & DTR Behavior on Reset**

In Figure 10 we see a view of a reset cycle where the problem manifests. Several things manifest here which provide clues to the final piece of the puzzle. First clue, notice that when **EN** goes LO, **IO_0** follows. This is a little curious since the reset circuit can not pull both signals low at the same time. The reason **IO_0** goes LO is that when the ESP32 is in Reset state (**EN** = LO), **IO_0** is configured to be an input signal. With no external pull up this node is discharged and also goes LO.

Second clue, notice that when DTR goes to 0, EN immediately starts to rise. However, for some reason, a short time later (a little more than 1ms) IO_0 starts to rise, even though our control signals have not changed. Let's take a closer look at this.

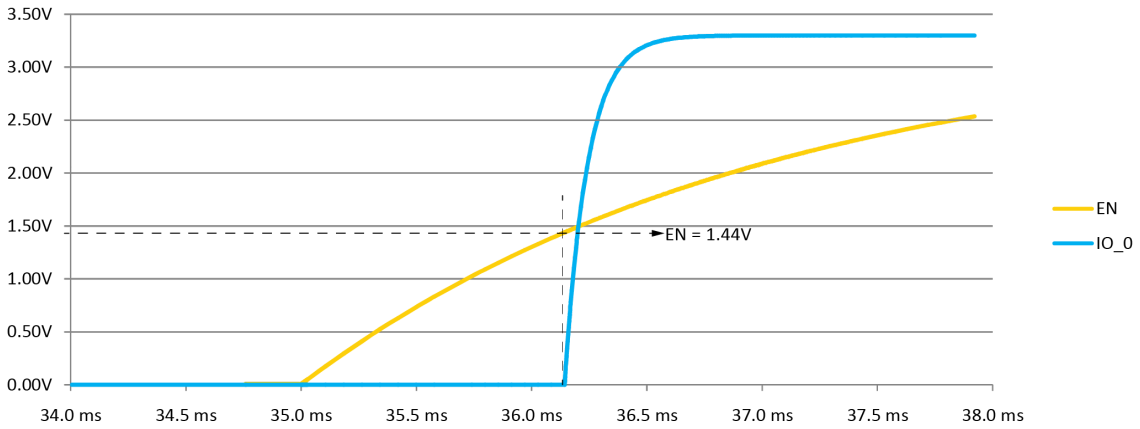



**Figure 11: Zoom-in of EN vs. IO_0 Behavior on Reset**

As we see in Figure 11, **EN** starts rising at 35ms, and at 36.15ms **IO_0** starts to rapidly rise. What is going on here is that despite the indications in the ESP32 datasheet about the $\mathbf{V_{IL\_nRST}}$ being 0.6V, and $\mathbf{V_{IH}}$ = 2.45V, the ESP32 starts to "do stuff" when **EN** reaches 1.44V… neither HI, nor LO. In this case, it turns on the internal pull-up resistors on **IO_0** in preparation to check whether to start up the Application or Bootloader. Since we know the only way our circuit can allow both signals to rise is if **DTR** & **RTS** are the same value, and in fact we now know that for a short time during the reset sequence (between Steps 3 & 4), both **DTR** & **RTS** ***are*** set to 0, this explains why both signals can rise at this moment.

Studying the timing of the signals in **Figure 11**, we can see that if the delay between Steps 3 & 4 of the reset sequence (while **DTR=RTS**=0) is less than about 1ms, by the time the **EN** signal reaches 1.44V, **RTS** is set to 1, and **IO_0** is forced LO. If that delay is longer, then **IO_0** can jump up for a short time. Now also consider Figure 6, where we show that adding a 1uF capacitor onto **EN** causes this signal to rise much more slowly. It follows then that we mitigate the problem by causing the **EN** signal to take longer to reach 1.44V, and therefore can tolerate a longer delay between Steps 3 & 4. However, as we also discussed, slowing down **EN** could have other undesirable consequences.

To illustrate this idea, **Figure 12** shows the same circuit, but this time with a delay between Steps 3 & 4 of the sequence set to about 1ms. We can see that in this case, **IO_0** never rises because it is forced low before the ESP32 internal pull-up resistors have been activated.

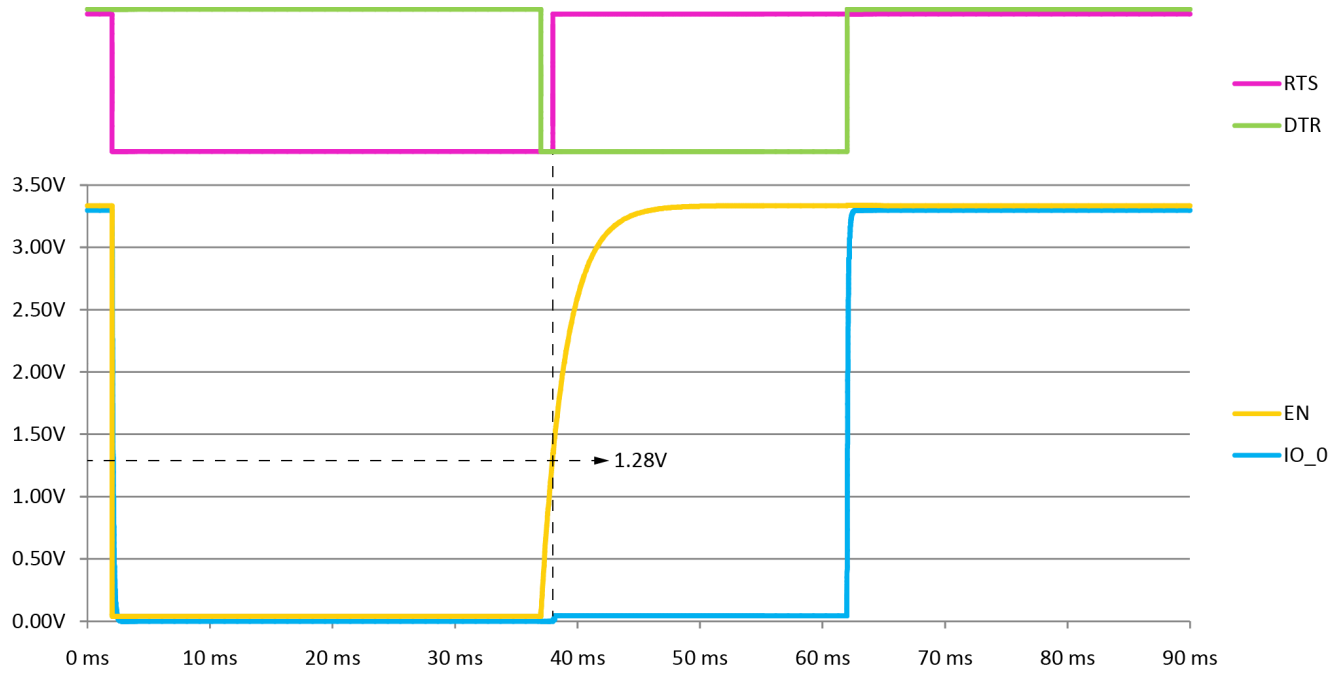



**Figure 12: EN vs. IO_0 Behavior with Small DTR-RTS Delay**

As we can see in **Figure 12**, the problem goes away if we can get **DTR** & **RTS** to transition as close to simultaneously, or at least within 1ms of each other, during the reset cycle. Let's examine how we can accomplish this with the modified reset circuit.

# Bringing Things Together

With an understanding of the problem and how the stock reset circuit works, let's examine how our modified reset circuit can solve this problem. For this, we will now analyze how the modified circuit behaves during that critical transitional delay between **DTR** & **RTS** changing state.

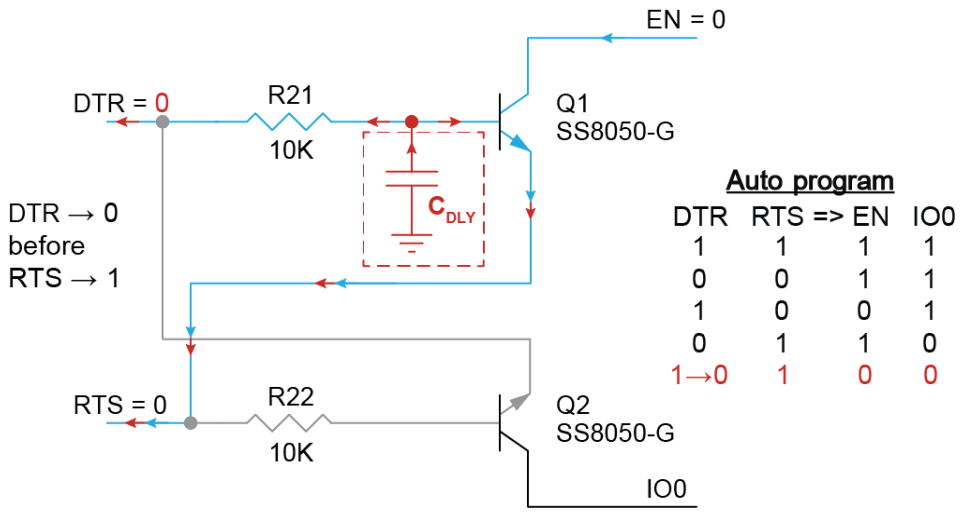



**Figure 13: Modified Reset Circuit Behavior During Transition**

Here in **Figure 13** we see some similarities to **Figure 8**, but with a twist. Just as in **Figure 8**, the Blue arrows indicate primary current flow for the **EN** signal, but now we also have Red arrows indicating current being delivered by the capacitor $\mathbf{C_{DLY}}$ connected to the base of Q1. Because *both* **DTR** and **RTS** are 0 at this time, current flows from the capacitor through R21 towards **DTR**, but also through Q1 towards **RTS**. $\mathbf{C_{DLY}}$ stores energy during the first part of the reset sequence when **DTR** = 1, then when **DTR** = 0, it delivers that energy back to the circuit. In this case, it means keeping Q1 turned on for a while after **DTR** transitions to 0. This, in turn, holds **EN** LO for long enough for the Host OS to command **RTS** to transition to 1. Once RTS = 1, we can see from Figure 8 that Q1 is immediately turned off, and Q2 is immediately turned on. This has the effect of simultaneous transition of DTR & RTS to their opposite states. As we saw in the previous section, those are the ideal conditions for the reset circuit and IO_0 is forced LO by Q2 long before EN can reach the 1.44V threshold for turning on the ESP32 internal pull-ups.

For this to work, $\mathbf{C_{DLY}}$ must be large enough to hold sufficient energy to keep Q1 saturated, **EN** = LO, for the longest expected delay between the control signal transition. In simulation we have found that 22uF works well for about a 10ms gap. In practice, most ceramic capacitors have a large tolerance of capacitance value, so in practice we have selected a 33uF capacitor. There is no practical upper limit on the value of $\mathbf{C_{DLY}}$ except size and cost. As soon as **RTS** is set to 1, Q1 turns off regardless of how much energy remains in $\mathbf{C_{DLY}}$ and the rest is sent into the **DTR** signal. Because the "idle" case (between reset cycles) is for DTR = 1, the capacitor has sufficient time to charge.

To see this solution in action, we simulate the same circuit as was used in **Figure 10**, but this time with a 33uF $\mathbf{C_{DLY}}$ in place and a 10ms gap between the **DTR** & **RTS** transitions. Now we see the timing of EN going HI aligned with **RTS** and no momentary pulse of **IO_0** during the sequence. The ESP32 samples **IO_0** as LO and initiates the Bootloader.

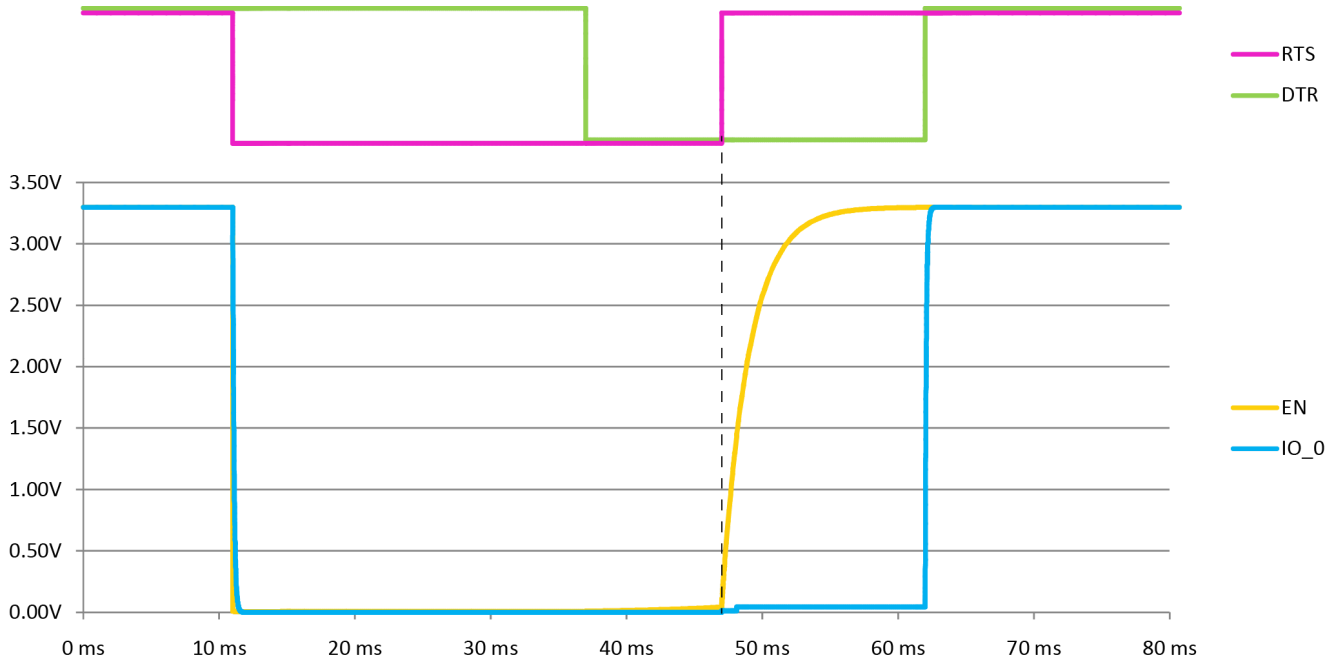



**Figure 14: Timing Behavior of Modified Reset Circuit**

As we see in **Figure 14**, this circuit is resilient across a variety of **DTR-RTS** timing gaps with a sufficiently large capacitor. The circuit also exhibits best-case **EN** transition timing vs. loading **EN** with a capacitor. The timing is now always aligned with the **RTS** transition.

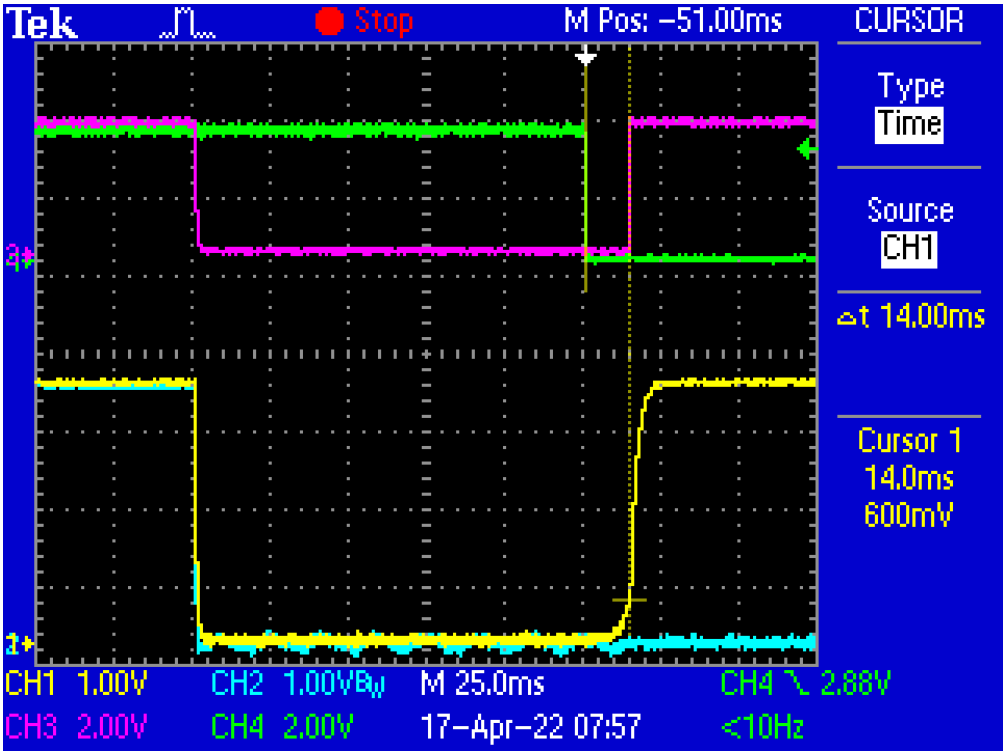



**Figure 15: Scope Capture of Modified Reset Circuit in Action**

We measured actual performance of this modification on an ESP32-DevKitC-v4 and include a scope capture in **Figure 15** of espflash attempting to reset the DevKit and download firmware. In this case, the **DTR-RTS** delay was measured to be about 14ms, and we have about 24uF (measured) of capacitance for $\mathbf{C_{DLY}}$ in the circuit. In the scope image we can see that **EN** rising is aligned with the rising edge of **RTS** and that **IO_0** exhibits no positive spike. Also notice that **EN** has a good rise-time, consistent with the simulated results. The download was successful for this run!

# Conclusion

We hope that this explanation of the little 2-transistor and 2-resistor circuit was entertaining and informative. It is our hope that the community will evaluate this simple modification to the existing reset circuit used on the ESP32 DevKits and consider integrating such an adjustment into future revisions of the board. Also, as seen in the pictures above, it's not hard to hack this solution onto your existing DevKit. If you do, you can eliminate that pesky capacitor on the EN line and enhance the reliability of your system!

One final note: One of the folks who reviewed this pointed out that, in fact, the transistors are probably "underbiased" to keep these particular transistors saturated, even under normal conditions. The circuit designers may also find value in taking a closer look there to improve the robustness of the base circuit!

Enjoy!