IoT in the Matrix: Mocking External Dependencies for IoT Systems Using Physical Hardware

Bartholomew Hsu, Jovan Araiza, Jeff Borg

bart.hsu@outlook.com, araiza1852@gmail.com, jeff.borg8@gmail.com

Abstract

Using software mocks as stand-ins for external dependencies is a well-established practice for software testing. However, mocking subsystems in IoT (Internet of Things) devices that interface with the physical world often isn't practical because the behavior of those subsystems is too complex or opaque. This often necessitates resorting to use of lab test instruments by specialists or field testing with real hardware, which can be difficult and expensive. However, quite a lot of advancements have occurred in hobbyist test instruments, electronics modules, and open-source software over the past couple of years; so that with a little bit of hardware savvy and not too much cost, simulating some of these interfaces has become viable for teams with limited resources. Our team has been able to apply this, using COTS (**commercial-off-the-shelf**) hardware to mock most of the physical-world interfaces of a IoT device such that it's unaware that it's not connected to reality. In this talk, we cover some of the background hardware knowledge necessary, how it can be applied to simulating some of the more complex interfaces, e.g., Wi-fi, global navigation satellite constellations like GPS, power sources etc., costs, advantages and limitations of going with hardware-based mocking, and some of the difficulties we encountered along the way.

Biography

Bartholomew Hsu is an experienced tester who specializes in IoT embedded devices and network services. He is currently with Impinj Inc. performing testing for UHF RFID readers and gateways. He received his education in computer engineering from the University of Michigan.

Jovan Araiza is a software engineer with 4 years of experience testing IoT software and hardware at Impinj. He has experience testing every layer from web UI down to embedded microcontrollers in IoT devices. He graduated with a bachelor's degree in computer science from Washington State University in 2019.

Jeff Borg currently works as an SDET testing IoT firmware. His bachelors' degree (1994) is in Electrical Engineering and his experience includes 20+ years of board level circuit design, including RF circuits. Combining this with significant software coursework at University of Washington, Jeff contributes in an environment where both disciplines are useful.

1 Introduction

IoT (Internet of Things) and embedded / mobile devices have subsystems that interface with the physical world, making them difficult for software testers to work with. Simple mocks do not adequately capture behavior and simulating inputs to these systems conventionally requires expensive electronic test instruments and specialists with knowledge to operate them. Software testers were faced with being unable to adequately cover device behaviors that relate to these subsystems.

But that's much less true today than it was ten years ago. Advancements in electronics have resulted in test instruments and electronic equipment that are nearly as capable as their professional equivalents but priced within reach of hobbyists and enthusiasts. Combined with this, a resurgence of public interest in hobbyist electronics and radio has in turn resulted in an increase in both documentation / tutorials and software tools that make use of these devices.

While a bit of self-study and trial and error may be involved, it is within reach of software testers to make use of these resources. Testers can set up hardware and software to mock out these interfaces with enough capabilities to be able to cover important test cases that weren't practical to cover before and at low enough cost to be acceptable. This approach is different from what electrical engineers might suggest but their context and the equipment available to them are much different from that of a software test team.

Our goal is to provide examples that can be set up locally or at least provide insight into the right questions to ask of your colleagues who work with hardware to help you solve the testing problems that you face.

2 Wi-Fi Mocking

2.1 Context And Motivation

Wi-fi needs barely any introduction, having become ubiquitous as the networking mechanism for IoT / embedded systems where wired networking is unavailable. Our interest as testers is to verify interoperability for devices with wi-fi networks at a functional level. However, even without digging into the RF aspects, wi-fi has many configuration settings, each with its own boundary conditions to probe: SSIDs (which may be ASCII, UTF-8 or just random bytes and hidden or not), PSKs/passphrases, authentication methods like WPA-PSK or the various flavors of WPA-EAP, frequency ranges, and so forth.

Finding a way to do this testing manageably is challenging. Manually reconfiguring a consumer-grade wi-fi access point for scenarios would be time consuming and tedious. Business-grade access points are costly and often come with SaaS for remote management that adds to the cost and complexity. In either case, having a wi-fi access point maintained by the QA or development teams that is connected to the corporate network can raise significant security concerns as well as concerns about wi-fi congestion from the IT department. One might attempt to ask their IT department to provide various configured wi-fi networks instead, but they are not really equipped to provide large numbers of wi-fi networks and would still have the security and congestion concerns.

What we're looking for is a solution that avoids these issues while retaining the capability to freely change the configuration. We can do that by looking for a solution that's somewhat off the beaten path. By using stock Linux packages that implement the functionality of a wi-fi access point (not everything but sufficient for software testing purposes) and some application of basic radio frequency (RF) principles, we can configure a Raspberry Pi 4 with a USB wi-fi adapter to act as a simple wi-fi access point. This then can be directly connected via an RF cable to the wi-fi antenna port of the device we are testing. Creating a wi-fi network for the most common configurations can be done on the fly by revising the settings and restarting the appropriate Linux service. Security concerns are reduced because the direct RF cable connection drastically shrinks the range at which an attacker can attempt to make a connection to the test wi-fi network, wi-fi congestion concerns are also reduced for the same reason, and cost is low because of the off-the-shelf parts used.

2.2 Hardware

Before diving in, we need to digress briefly to explain some basics of working with radio signals which, in turn, explain how to make hardware choices that achieve the desired goal. What we seek is:

- The right combination of cabling and connectors to create a cabled RF connection to pass wi-fi signals between the test wi-fi access point (WAP) and the device under test (DUT).
- To pass the correct range of frequencies between the test WAP and DUT. This is primarily a matter of identifying the frequencies that we are working with and then selecting components that are rated by the manufacturer to behave correctly at the frequencies that we need.
- To pass the wi-fi signal at the correct power level between the test WAP and DUT. A signal at a power level that is too high will damage the receiver circuitry on the other end and can be remedied by reducing the strength of the signal using a device referred to as a RF attenuator. A signal that is at a power level that is too low will not be detected by the receiver and can be remedied by amplification. The goal is to strike a happy medium where the signal strength is just barely powerful to ensure good connectivity.
- To be aware of RF leakage. Much like deep bass notes in music penetrating a wall, radio waves can be difficult to keep contained. Even with the use of a RF cable to contain the wi-fi signal, you may find that the wi-fi network provided by the test WAP is visible from nearby laptops and phones. This may require you to relocate your test rig to prevent attempts at unauthorized wi-fi access.

To accomplish those things:

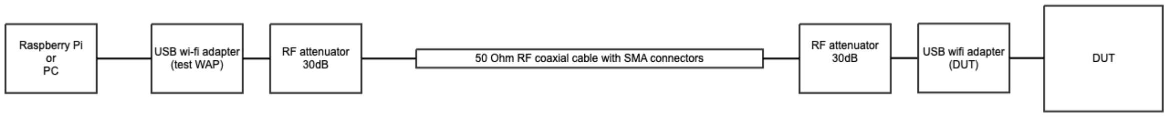
- From the test WAP side, the need to create a cabled connection means selecting a USB wi-fi adapter that has a threaded external connector (RP-SMA) for an antenna to which a cable can be attached. Similarly, from the DUT side, a corresponding means of making an external connection is needed. Many IoT/embedded devices either directly provide an RP-SMA connector or a tiny u.FL connector for wi-fi. For our scenario, we were fortunate that wi-fi support for our DUT was through USB wi-fi adapters with certain chipsets and we were able to procure models that had the RP-SMA connector. Because the components below use normal SMA connectors, a pair of RP-SMA to SMA adapters will also be needed.
    - As an option of last resort for DUTs that do not have an external antenna connector, one might consider asking for a DUT to be modified to replace its onboard antenna with a connector, either u.FL or RP-SMA.
- Identifying the frequency range can be done in a few moments. Wikipedia tells us that the frequency ranges used for the most common wi-fi protocol versions (802.11n, -ac, and -ax) are 2.401-2.495 GHz and 5.150-5.895 GHz. For the relatively short distances (<1m) and the moderate (<6GHz) frequencies involved, 50Ω cables with SMA connectors are inexpensively available. Other components, such as the attenuators discussed next, should also be chosen that have a rating for up to 6 GHz.
- For managing power levels of the wi-fi signal, if we turn to any convenient search engine, we find that the maximum power output allowed for a WAP is 1 Watt (4 Watts for some channels in some jurisdictions but not the one the authors are in) which translates to 30 dBm (decibels relative to one milliwatt). We also find that the signal strength necessary for receiving a wi-fi signal ranges from -30 dBm to -70 dBm. This means that, in the worst case, we must attenuate the wi-fi signal by a factor of at least 60 dB (10^6) to go from +30 dBm to -30 dBm and still have a safety margin of 40 dB (10^4) if the actual transmit power is not at the maximum 30 dBm, which it most likely isn't.

As it turns out, 60 dB of attenuation in the form of two 30 dB attenuators rated up to 6 GHz attached at each end of the RF cable is sufficient to maintain good signal level.

  - It's also worth noting that RF attenuators have a rating for the power that they can safely dissipate that is separate from their attenuation value. In the context of wi-fi, the common 0.5 W rating is sufficient but when working with stronger signals, an attenuator with an appropriate power rating needs to be chose to avoid damage from overheating.

- For minimizing RF leakage, the least costly and most effective measure will be to use as short a RF cable as reasonably possible to reduce emissions from the cable. Another option is to restrict the maximum transmit power level allowed for the wi-fi adapter from the OS but not all wi-fi chipset drivers support this.

The resulting hardware will look something like



2.3 Software

With the hardware side sorted out, we turn our attention to the software side of our test wireless access point. We are using a Raspberry Pi 4B running the current version of Raspberry Pi OS based on Debian 12 ("Bookworm"). We want the DUT to be accessible and assigned an IP on the same network that the Raspberry Pi acting as our test WAP is on instead of a linked but isolated network, as is the default with consumer wi-fi routers one might find at home. In networking parlance, we want the test WAP to act as a network bridge instead of a router. How might we do this?

The first and simplest of our two options is to use the built in functionality of the Network Manager package, supplied and enabled by Raspberry Pi OS by default, that is intended to allow users to set up a temporary wi-fi hotspot. The steps are provided and supported by Raspberry Pi's official documentation ("Use your Raspberry Pi as a network bridge") so there is no need to describe them here. A limitation is that it supports only WPA-PSK authentication, commonly found on consumer wi-fi routers where a single passphrase is used by all devices.

The latter option allows much more configurability but is also much more complex: hostapd. It is also supported on Raspberry Pi OS and was the recommended mechanism for configuring a Raspberry Pi as a WAP in previous releases before the inclusion of Network Manager. Set-up is a bit more complex: a network bridge with an ethernet interface needs to be created through Network Manager. Network Manager needs to be configured to release control of the network interface corresponding to the USB wi-fi adapter so that it can be controlled by hostapd, and then hostapd needs to be installed and configured. Working out the proper configuration is not simple but, for our efforts, we are rewarded with being able to create configurations that match many more common scenarios, including support for the WPA-EAP authentication and its various encapsulated protocols like EAP-TLS and PEAP-MSCHAPv2, both commonly used on corporate networks, to enable multiple authentication mechanisms on the same network.

3 GNSS ("GPS") Mocking

3.1 Context And Motivation

The backbone of modern location, navigation, and timekeeping are the several Global Navigation Satellite Systems (GNSS), constellations of satellites broadcasting signals that can be correlated by a receiver to identify its location with high accuracy. Examples include the Global Positioning System, a.k.a. GPS (US), Galileo (EU), BeiDou (China), and GLONASS (Russia).

Mocking out a GNSS signal might be necessary when:

- The GNSS signals are simply unavailable because they are unable to penetrate the building where the DUT is. This is the case for most office buildings.
- There may be a need to check behaviors related to the device being at a different location than its current physical location.
- There may be a need to check behaviors related to large scale movement of the device.

Some buildings have or can be retrofitted with a rooftop GNSS antenna cabled into the building for timekeeping purposes, e.g. for NTP servers or for GPSDOs (high precision oscillators used for electronics testing that use GNSS timekeeping data to stabilize and increase their accuracy) but such antennas are costly to install if not already present or may be in places not accessible for testing purposes. The simple expedient of physically moving the DUT close to window may not always be an option and only provides a half-view of the sky, affecting accuracy and other behavior. The orthodox solution would be to use specialized test instruments to simulate GNSS signals, including multiple constellations at once and movement. But those devices are incredibly costly and are intended for specialists evaluating receivers at a very deep level.

As before, let’s look at something unorthodox: there are hobbyist open-source software packages for use with software defined radios that simulate signals from the United States’ GPS satellite constellation. The capabilities are limited but sufficient for a GNSS receiver to get a position fix and therefore allow us to execute test scenarios.

3.2 Hardware

With continued drops in the cost of processing power, a new way of working with radio signals was developed: the software defined radio (SDR). At a conceptual level, the circuits in any RF system are mathematically transforming the radio signal in stages to get to a final result. An SDR based system instead treats the signal as digital data and processes it using code, with the RF circuitry serving only to convert a received radio signal to binary data for processing or to convert processed binary data to a signal and transmit it as a radio signal. As costs continued to drop, SDRs capable of both receiving and transmitting signals came in reach of people at a hobbyist / HAM radio level, which means more software became available.

The software package we are using, gps-sdr-sim, includes a list of SDR models it can be used with. While neither they nor we endorse any specific device, we will note that more expensive SDRs from makers of test instruments (e.g. the Ettus B200 from National Instruments) come with superior documentation, tooling, and support by software packages. Less expensive hobbyist SDRs cost less than a third of the price but may require more expertise and effort to get working. Which trade off you make will necessarily depend on your circumstances.

(At this point, we are obligated to digress momentarily into legal and regulatory matters. You may see tutorials and video demos where a person hooks up a SDR to an antenna and transmits over-the-air to demonstrate simulating GPS. You may be tempted to do the same if your DUT does not have a GNSS antenna port. <u>These demos are almost certainly being performed illegally.</u> Readers should be aware that, in nearly every jurisdiction globally that

- transmitting a radio signal requires compliance with regulations and appropriate licensing
- that a signal being transmitted is easy to track down with the right equipment
- that the fines and penalties for those caught can be draconian

In short, never use a SDR to transmit using an antenna unless you are very sure that you are doing so in compliance with the laws of your jurisdiction.)

Going back to our RF basics discussed in the section on wi-fi:

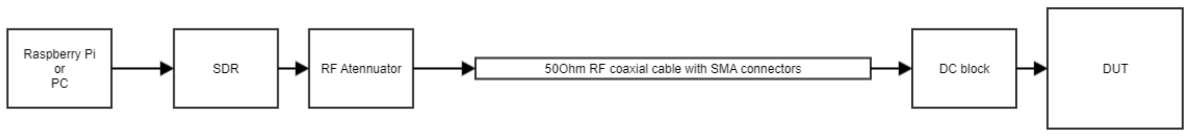
- A brief visit to any search engine reveals that the primary frequency range (the L1 band) used by GPS is centered at 1575.42 MHz with most other GNSS constellations having their L1 frequency either at the same frequency or nearby. While you should still check the frequency ratings for cables and other components, this frequency is low enough that most components should be capable of handling it without any problem.
- Again turning to a search engine, we find that the signal strength that would constitute a sufficient signal is -125 dBm to -155 dBm. (These are fantastically small quantities: $10^{-15.5}$ to $10^{-18.5}$ watts which highlights what an engineering marvel GNSS systems are.) This means that we will need attenuators again but how many depends on the signal strength being put out by the SDR. For the moment, let us stipulate that attenuators will be needed and that we will need to take a measurement of the output to determine exactly how much is needed.
- At such low signal levels, leakage is not an issue.

To those, we add one other consideration:

- Because of the extremely weak signals involved, GNSS receivers are often operated with antennas with built-in amplifiers. These amplified antennas require power so the receiver will supply a voltage through the antenna cable to the antenna to power it. Needless to say, a SDR's transmitter output is not designed with the presence of this voltage in mind and there is potential for it to be damaged. Therefore, one additional component is needed: a DC block, which, as its name suggests, will block the unwanted fixed voltage supplied by your GNSS receiver while still letting the RF signal pass through. Forgetting to install the DC block may result in damage to the SDR.

We still need to determine the base output power of a signal transmitted by our SDR of choice so that we can determine how much attenuation to use. To do that, we need to introduce one other test instrument, the spectrum analyzer, which, to oversimplify, when tuned to a particular frequency range, will show how much power in dBm is detected in that frequency range as a graph. (If one is not available, we can again turn to hobbyist grade instruments: the TinySA Ultra is well regarded in HAM radio circles, is very inexpensive, and has been used by us to take such measurements). The procedure is simply to connect the SDR's transmit output to the spectrum analyzer with an appropriate attenuator attached for safety, configure the SDR to transmit a sine wave at the L1 frequency, 1575.42 MHz, with zero amplification added by the SDR, configure the spectrum analyzer to examine the L1 frequency range, and read off the height of the peak, which will be the default dBm power. If the signal is not visible, reduce the size of the safety attenuator and try again. From that value, subtraction and then addition of the dB value of the safety attenuator will tell us how much additional attenuation is needed to get to roughly -125 dBm. Precision is not needed; to within +/- 10 dBm should be fine.

The resulting hardware looks something like



3.3 Software

On the software side, as mentioned earlier, we will be using an open-source package named gps-sdr-sim, which is available as a repository on GitHub. gps-sdr-sim does not operate in real time; instead, it generates a signal file representing the GPS signal and that file is played back through an SDR. To summarize the process:

- Choose the latitude, longitude, and height above sea level you wish to use for your simulated antenna location and specify it as a static location. (The gps-sdr-sim does provide support for simulated movement as well but that functionality has not yet been evaluated.)
- Choose a date and time and retrieve the appropriate GPS constellation ephemerides file from the NASA CDDIS site ("Daily RINEX V2 GPS Broadcast Ephemeris Files"). These files describe the location and movement of each of the GPS satellites in orbit, which determines the set of GPS satellites that are visible from the chosen location and what distance they are at. This is necessary to compute the RF signal that the GNSS receiver should be receiving at that location. Because each file covers a timespan of one day, the time from which the simulation can start is limited by the range of the file.
- Choose a duration. For doing GNSS testing, the concept of time to first fix (TTFF) and cold/warm/hot starts should be familiar. The duration chosen should be for the cold start TTFF plus the scenario duration.
- Issue the command to the GNSS receiver to reset to the cold state. Remember that the generated signal file isn't going to be synchronized with the actual current time. Having the receiver in the cold state ensures that it doesn't see an unexpected jump in time from any previous runs.
- Issue the command to begin playing back the RF signal file through the SDR.
- Begin observing the output of the GNSS receiver. This is usually through a serial connection exposed to the Linux OS as /dev/tty* outputting NMEA 0183 formatted data sentence in plain text. The GPGGV sentence output should begin to show satellites becoming visible as well as their apparent signal strength. You may need to increase the amplification level of the SDR playback command if the signal strength is low or no satellites are visible. After sufficient satellites are visible and at or before the cold start time has elapsed, the GPGGA sentence should be populated with the latitude and longitude that the receiver thinks it's at. Other NMEA 0183 sentences may be available, refer to the documentation of the receiver module being used.
- At this point, the GNSS receiver module has and should retain a fix until the playback ends and your tests can begin executing.

3.4 Learnings And Limitations

There are quite a few limitations:

- As mentioned before, this simulation isn't synchronized with the actual time. This does not pose an issue for the authors' test scenarios but may pose a problem for other scenarios.
- Simulation time can be constrained on low end computers like the Raspberry Pi 4 by the large file size of the generated RF signal, roughly 3 GB for 5 minutes worth of signal data, and by computation time, nearly 1 second of CPU time to generate 1 second of signal data.
- This simulation has not been evaluated against receivers with Assisted GPS (A-GPS) enabled. Such systems will have satellite ephemerides data for the current time, restricting the start time for the simulation to near the current time.

These limitations, while considerable, need to be weighed against the cost savings.

4 Battery / Power Mocking

4.1 Context And Motivation

Maximizing battery life is crucial for mobile devices and battery powered IoT applications and part of that is being able to assess power consumption in varying test scenarios. Some devices, like smartphones / tablets, have a built-in battery fuel gauge chip that can provide this data for you but not every DUT includes this kind of circuitry. In that situation, we can look instead to lower cost power monitoring test instruments instead that will effectively be a mock taking the place of an actual battery.

4.2 Hardware + Software

To provide the necessary background in brief, a battery contains a certain quantity of energy that it can deliver, much like fuel in a fuel tank, before becoming fully discharged. That total energy is measured in terms of power delivered over a duration: watt-hours. The rate at which energy is delivered from the battery varies depending on what the DUT is doing, so measuring power consumption boils down to measuring the current and voltage at very short intervals, computing the power from that, and adding together these myriad measurements together to determine the total energy consumed. The measurement frequency affects the final accuracy as well; measurements taken at a high rate are less likely to miss transient spikes in power usage.

Fortunately, test instruments exist that do the job are available at a manageable cost compared to full sized DC benchtop power analyzers. For our test configuration, the Monsoon Solutions High Voltage Power Monitor (Model AAA10), is a combined power supply and high precision power measurement tool that meets the voltage and current requirements for the products we are testing. (For handheld/mobile devices that use batteries that deliver <4.5 V, there is a corresponding Low Voltage Power Monitor (Model FTA22D) that is readily available on the secondhand test instrument market.) The manufacturer provides Python and other libraries for configuring the instrument and taking measurements. For test configurations that have higher power requirements and where therefore less precision is needed, benchtop power supplies with USB/network interfaces that allow power readings to be logged to a computer may be a viable lower cost option.

4.3 Testing process

The test process is:

- Configure the voltage to match the requirements of the DUT
- Boot the DUT
- Set the DUT into the desired state for which power measurements are to be made
- Begin logging power measurements through the power measurement tool's API
- Allow the DUT to execute the scenario
- Perform analysis of the gathered data, including computing the average power consumed and checking the data points for unexpected bumps/spikes in power consumption

Possibly the biggest difficulty with making power measurements is repeatability, particularly for handheld devices which necessarily have small batteries and are therefore the most severely power constrained. Because any modern IoT device is a veritable hive of processes and subsystems with each conducting activity on its own schedule, if you graph power consumption versus time, you will see many transient spikes caused by that activity. It is unfortunately effectively impossible to trace those transient events to the activity that originated it. Adding instrumentation and logging to the code to attempt that would itself trigger additional activity that adds more noise to the data, defeating its own purpose. Ultimately, we need to accept that making power measurements requires us to deal with some level of randomness which requires use of statistical principles to tame. Some tips:

- Putting the DUT into as quiescent state as possible helps improve reproducibility. Disabling services / software and turning off any hardware subsystems not relevant to the scenario being executed can help eliminate noise in the measurement. This has to be balanced against the risk of turning off too many things and causing behavior that doesn't reflect what the end-user will see.
- For average power measurements, increasing the duration of the measurement can reduce the effect of noise on the average. For example, a one-off 1 second burst of activity can significantly skew the average power measured over a 1 minute duration but has very little effect on the average power measured if the measurement duration is instead 1 hour.
- The inverse works as well: one can amplify the effect of an action that individually has very little power consumption, enough to make it measurable, by repeating it many times over a long duration. Assuming that you have a baseline average power measurement that's repeatable, you

can take an average power measurement while repeatedly triggering an action N times and then subtract that average power measurement from the baseline average power measurement and divide it by N to get an estimate of the power consumption for that action.

5 Mocking other interfaces

5.1 Mocking for GPIOs and other digital I/O

Industrial embedded systems and IoT devices often have the capability to interact with external hardware through GPIO (General Purpose Input/Output) pins that can be configured through software to act as either input signals or output signals, e.g. handling buttons or other user inputs or triggering buzzers / lights for the device being controlled.

For testing purposes, probably the most convenient and easily accessible way of mocking out a GPIO interface is the Raspberry Pi, which itself has a set of GPIO pins and well documented and supported libraries to configure them, making it relatively straightforward to mock out any device one would connect to the DUT's GPIO. In the authors' case, the Raspberry Pi's GPIO were configurable to be directly compatible with the DUT, but in some cases, you may need an additional circuit to translate from one interface to another.

5.2 Mocking options for RF signals

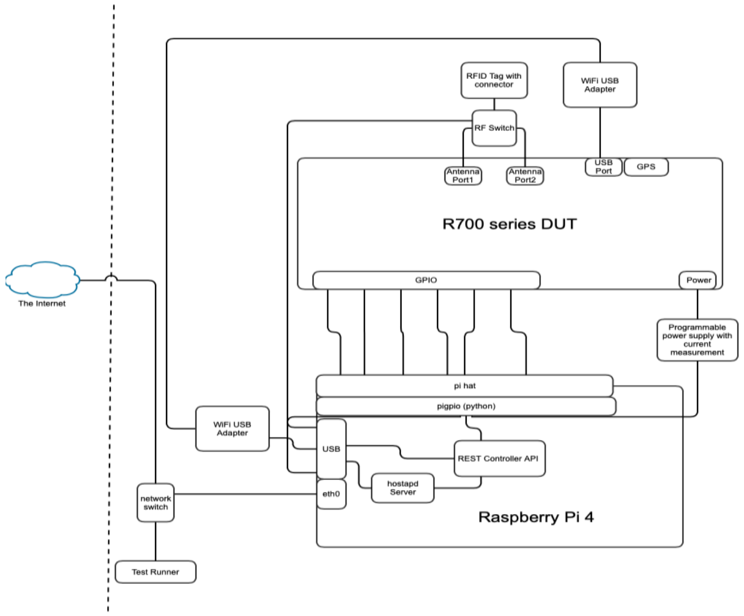
We've talked about the RF signals for wi-fi and GNSS in preceding sections but there are IoT devices that work with other types of RF signals, e.g. the authors work with UHF RFID tag readers operating in the ISM frequency range. There are a couple of other things that software testers can deploy to manipulate RF signals for mocking purposes: USB RF switches and USB RF attenuators.

A USB-controlled RF switch is a bi-directional device that can be used to switch a single "common" RF port to connect to one of a set of RF ports by issuing commands to it over USB. The manufacturers provide libraries and sample code to do so, although the quality may be variable. As an example of how it might be used, a number of the authors' test scenarios involve a UHF RFID tag wired up to a SMA connector (instead of its usual antenna) connected to the common port of a USB RF switch and with the remaining switched ports connected to the antenna inputs of the RFID reader. With this arrangement, we can make the tag seem to be visible from any of the RFID reader's antenna ports by simply issuing a command to the switch.

Though not currently used by the authors in any test scenarios, another device worth being aware of that apply to test scenarios are USB-controlled RF attenuators which provide programmable attenuation to signals passing through the device.

6 Overall system

A system diagram is presented below with all of the mocked out interfaces in place. The DUT is operating as it normally would, unaware that its hardware interfaces are not connected to the real world.



To provide control over these mocked out interfaces, we created a management REST API that is served through a Python web server. Tests that are executing can simply make the appropriate REST calls at the appropriate time to set the configuration needed for the current test scenario.

7 Conclusion

In conclusion, we hope we have shown that advances in hobbyist-grade electronic tools and less well-known electronic test instruments have put mocking out hardware interfaces for the mobile devices and IoT systems they within the practical reach of software testers. While it is true that a bit of electronics and other knowledge is needed to take advantage of this, most of the knowledge is terminology and various rules of thumb rather than anything complex.

8 Disclaimers

The views expressed in this paper and presentation are those of the authors, not Impinj Corporation.

9 Acknowledgements

The authors gratefully acknowledge the support provided by Impinj for this paper as well as the support of colleagues there who have provided review and feedback to improve it.