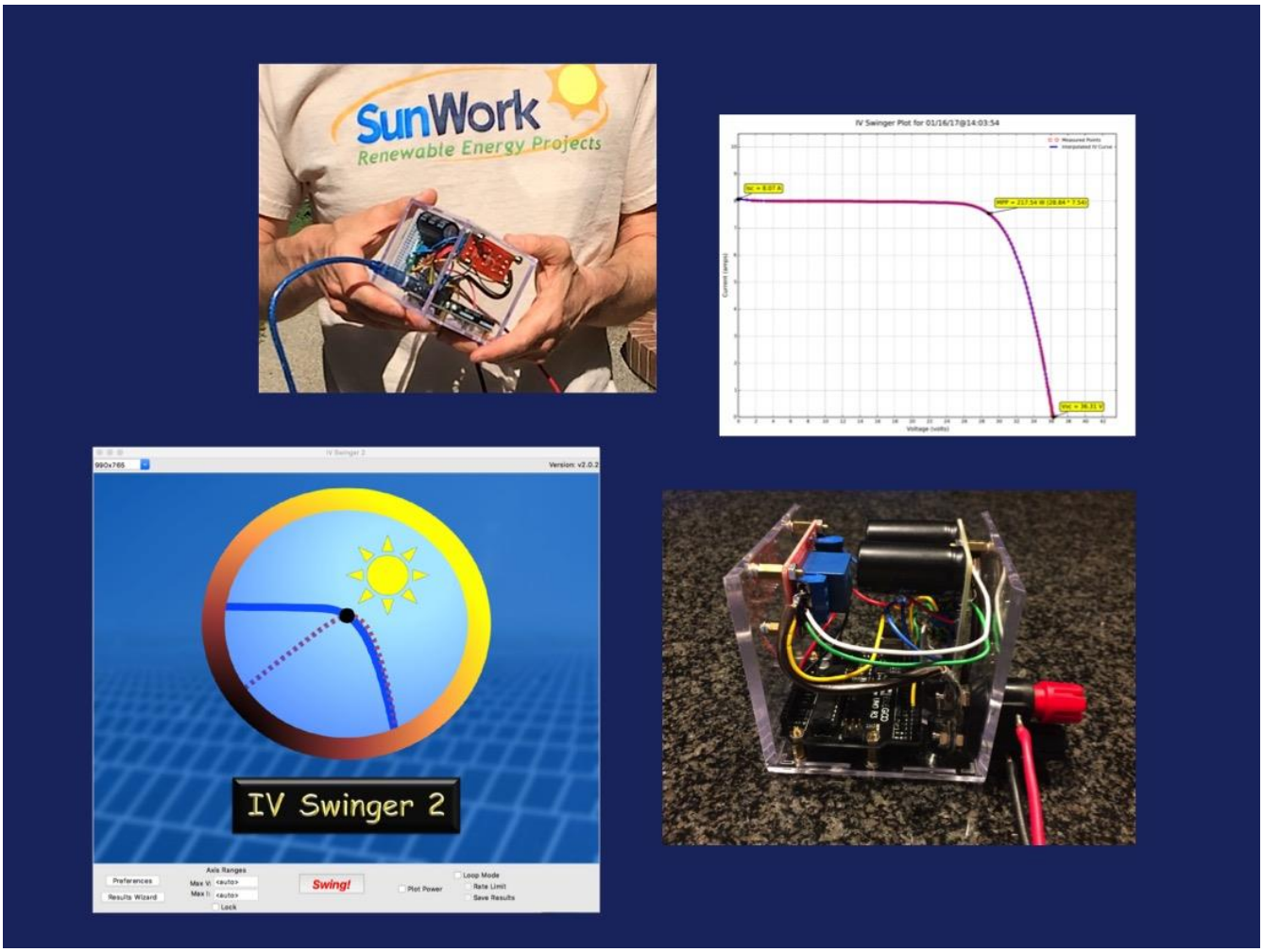
IV Swinger 2

Design and Theory of Operation

Document Revision: 1.2 (13-Jan, 2021) Chris Satterlee



IV Swinger and IV Swinger 2 are open source hardware and software projects.

Current versions of the license files, documentation, hardware design files, and software can be found at:

https://github.com/csatt/IV_Swinger

Table of Contents

Table of Figures

Table of Equations

1 Introduction

This document contains a detailed description of the design of the IV Swinger 2 (IVS2) hardware and software. It is not necessary to read this document in order to build or use the IV Swinger 2[1]. The intended audience is anyone who wants to understand the details of the design before, during or after building one.

It is assumed that the reader is already familiar with what the IV Swinger 2 is and what it is used for. At a minimum, the reader should have read the "IV Swinger 2 User Guide" before reading this document. It is recommended that the reader has explored the build instructions, photos, and other information on the Instructables website:

https://www.instructables.com/id/IV-Swinger-2-a-50-IV-Curve-Tracer/

Some degree of understanding of electronics and programming are necessary to understand everything in this document. However, an attempt has been made to explain the concepts and provide external references (e.g. Wikipedia pages), so readers with minimal technical backgrounds can understand as much as possible. Of course, that means that experienced readers may find the document pedantic and lengthy – sorry!

In order not to clutter up the document with URLs to the external references, they are mostly hyperlinks. That won't help if you are reading a paper copy, so save a tree and read the document PDF on a device with an internet connection that will allow you to click on the hyperlinks to open the external references in your browser. Internal cross-references are also hyperlinked in the PDF, so that is another reason not to read a paper copy. Some of the more important external references are listed with their URLs in the References section on page 189.

The predecessor of IV Swinger 2 was simply called IV Swinger, but will be referred to as IV Swinger 1 (or just IVS1) in this document. Only one IVS1 was ever built. The document "IV Swinger: Design, Construction, and Operation" has some information in common with this document. That information is repeated so that it is not necessary for readers to refer to the older document. In some case, however, the older document goes into more depth, and the reader will be referred to it for further detail.

[1] In fact, many people have successfully built the IV Swinger 2 in the past 2+ years, and I am just now belatedly writing this document.

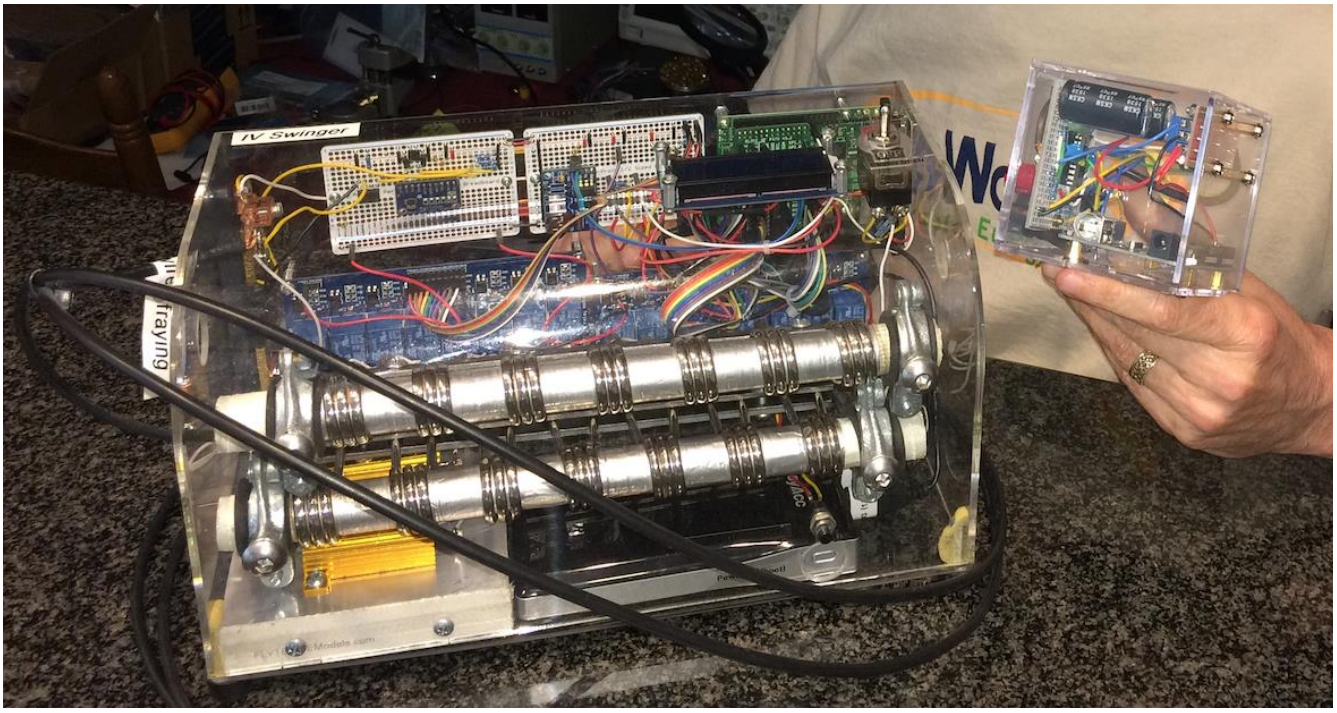


Figure 1-1: IVS1 and IVS2

1.1 GitHub Repository / Licensing

All of the IV Swinger (IVS1 and IVS2) code and documentation (including this document) are available in a public GitHub repository at https://github.com/csatt/IV_Swinger. You can use your web browser to look at the files and download them to your computer. It is better to install the GitHub Desktop app and use "Clone Repository" on the File menu, and then "csatt/IV_Swinger" on the URL tab. That will get you everything. If you prefer the command line, use:

```
git clone https://github.com/csatt/IV_Swinger.git
```

The IV Swinger is an open source hardware and software project. Permission to use the hardware design is granted under the terms of the TAPR Open Hardware License Version 1.0 (May 25, 2007) - http://www.tapr.org/OHL. Permission to use the software is granted under the terms of the GNU GPL v3 license - http://www.gnu.org/licenses. See the files in the GitHub repository for details.

1.2 Design Objectives

1.2.1 IV Swinger 1

The objectives of the **original IV Swinger design** were the following (copied from the "IV Swinger: Design, Construction, and Operation" document):

- **To be an educational tool**

The initial target use of the IV Swinger was for Gil Masters' "Electric Power: Renewables and Efficiency" (CEE 176B) course at Stanford. But any college-level (or possibly high-school level) course that covers photovoltaic IV curves could benefit from having an IV Swinger.

- **To be low cost**

Commercial IV curve tracers such as the Solmetric PVA-1000S cost over $5000. This is beyond the budget for most college courses. The objective for the IV Swinger was to have a total cost of parts in the low hundreds of dollars. The labor cost is assumed to be zero - anyone building one should be doing it for the fun of it (or possibly for academic credit).

- **To support a single modern PV solar panel**

Commercial IV curve tracers can handle the high voltage and power of a whole string of panels in series. This is not necessary for the experiments that are currently performed in an academic lab setting. The IV Swinger is designed to handle a single PV solar panel with $I_{SC} \leq 10A$, $V_{OC} \leq 80V$ and $P_{mpp} \leq 450W$.

- **To be portable**

IV curve tracing experiments are of course performed outdoors where the sun shines, possibly on a rooftop. The location may be out of the range of an extension cord. Therefore, a design goal was for the IV Swinger to be battery-powered and small and light enough to be carried by hand.

- **To be easy to use**

A typical student using the IV Swinger will only use it a small number of times, so a long learning curve would be counterproductive. An important design goal was to make it as simple and intuitive to use as possible.

- **To have the internals visible from outside**

The IV Swinger's transparent acrylic case exposes all of its innards to be seen by the user. This is more than just to make it look "cool" (which admittedly it does). It is also so it is not just a "black box" that magically spits out IV curves. Students who have traced an IV curve manually can pretty easily see that the IV Swinger is just a machine that automates the same process they did by hand.

1.2.2 IV Swinger 2

For the most part, all of the IV Swinger 1 objectives were carried forward or enhanced for IV Swinger 2. Here is the same list, with the differences between IVS1 and IVS2 highlighted.

- **To be an educational tool**

In addition to being an educational tool for students first learning about PV technology, IV Swinger 2 can be (and has been) used for graduate-level research projects. It has also been used for non-educational purposes; for example, matching salvaged PV modules to optimize string output.

- **To be low cost**

In comparison to the IV Swinger 1 cost of parts "in the low hundreds of dollars", the most expensive IV Swinger 2 variant costs less than $100 to build, and the least expensive costs less than $50. Minimizing the hardware cost was one of the highest priorities.

- **To support a single modern PV solar panel**

This is still a limitation of IV Swinger 2. However, the design is potentially scalable to work with strings of modules. Additionally, there are variants that work with a single PV cell.

- **To be portable**

IVS2 is much smaller than IVS1, although it does require a laptop computer. The IVS2 hardware is powered by the laptop, so there is still no need for AC power.

- **To be easy to use**

IVS2 is easier to use than IVS1 in almost every way. However, it also has many more features, so you might come to the opposite conclusion if you just look at the number of pages in the respective User Guide documents.

- **To have the internals visible from outside**

This is the only one where IVS2 falls short of IVS1. Although a transparent acrylic case is also used for IVS2, the mapping between the manual process of tracing an IV curve and the resistive loads and relays of IVS1 was much easier to relate to than it is to the IVS2 design.

Additional objectives for the IV Swinger 2 design were:

- **To be easy to build**

The #1 problem with IVS1 was that it was very labor-intensive to build. Only one was ever built, and the prospect of even documenting the step-by-step process to build one was so daunting that it was never undertaken.

The objective for IVS2 was for it to be possible for a first-timer to build in under 10 hours. The creation of PCBs was not an initial goal, but that has brought the build time significantly lower and reduced the possibility of mistakes.

Skill level and minimizing the requirement of special tools were also taken into account when making design choices. For example, only through-hole parts were considered since soldering surface-mount parts is much more difficult.

- **To instantly display the IV curve**

Using IVS1 was like using a film-based camera (or perhaps like a digital camera without a display). PDFs of the IV curves were saved to a USB flash drive for later viewing.

The objective for IVS2 was to be able to view each IV curve in near real time, and to be able to generate curves at a rate of one per second. This is tremendously useful when learning about the effects of shading.

- **To significantly improve the resolution**

IVS1 recorded approximately 20 points per curve. This produced surprisingly nice IV curves, especially when Catmull-Rom spline interpolation was used for smoothing. However, shading cases for PV modules with bypass diodes have very sharp inflections that were not resolved well with so few points. IVS2 can produce curves with well over 100 points, and the inflection points are nicely resolved. Even for cases where the shading is virtually undetectable by the eye, it is possible to see the very subtle "humps" in the curve.

- **To support calibration**

For the original target users of IV Swinger 2 (students first learning about PV technology), a high degree of accuracy is not critical. For some users, however, the ability to calibrate to a higher level of accuracy is desirable. IVS1 did not support any calibration. IVS2 supports both basic and advanced calibration.

- **To improve reliability and repairability**

IVS1 has many parts that can fail, and if they do fail, it is very difficult to access those parts for repair or replacement. An early objective for IVS2 was both to reduce the number of parts that are susceptible to failure and to make it very easy to replace any part that might fail.

- **To have more flexible options for PV connections**

IVS1 had cables with MC-4 connectors hardwired into it. Some PV modules have bare cables. Testing with a battery or bench power supply is sometimes desired.

IVS2 uses binding posts that accept either bare wires or banana plugs. This allows many different options to connect a PV module, battery, or power supply to the inputs.

- **To support both Mac and Windows laptops**

Once the decision was made to require a laptop for the control and display, it was important that the software be able to run on the two most common laptop platforms: Mac and Windows.

- **To support external sensors**

IVS1 had no support for external temperature and irradiance sensors. IVS2 does.

1.2.3 The Truth

My only original objective for IV Swinger 1 was "build something useful using a Raspberry Pi". Once the idea of building an IV curve tracer came to mind, I did write down a list of objectives that was pretty close to the list in Section 1.2.1 before starting.

The development of IV Swinger 2 was much less top-down and linear. Many of the "design objectives" listed in Section 1.2 above were not even something I considered until after some serendipitous breakthroughs. It is easy to write them down after the fact and make it seem as if I had a grand vision before starting and then executed it. The truth is much messier.

The "IV Swinger: Design, Construction, and Operation" document section entitled "Future Enhancements" lists only four ideas:

- Use 100W power resistors in place of immersion coils
- Use DC/DC converter for variable load
- Ruggedization
- On-board graphical display

The most interesting one to start working on was to use a DC/DC converter for the variable load. The hope was that this would allow much better resolution and would be smaller and easier to build. I spent a long time learning how to build a custom buck-boost converter (winding my own inductors, etc.) I did get this to work – sort of. But it was so noisy that the quality of the IV curve was much, much worse than IVS1. I would have had to solve that problem before that strategy would have been viable. I also had a problem with burning up components and was getting a bit discouraged.

It was at that point in time that I stumbled on Jason Alderman's blog post, "Wireless IV Curve Tracer for long term field testing", http://jalderman.org/?p=57. Using a capacitor for the load had not occurred to me before seeing Jason's design. Oddly, I had never actually researched different schemes for building IV curve tracers. Using a capacitive load was not Jason's idea, but his blog post was a critical turning point for IVS2, so I have to give him credit. I started experimenting with this scheme and the results were stunningly good. I never considered going back to the DC/DC converter scheme.

The capacitor scheme requires real-time processing, so a microcontroller such as the Arduino was needed (the Alderman design used a standalone ATMega644 microcontroller). I was still "stuck" in the mindset that the main software would run on a Raspberry Pi, and the Arduino would be controlled by the Raspberry Pi. Another critical turning point was when I realized that replacing the Raspberry Pi with a laptop made so much more sense. Figure 1-2 shows my actual notes from that "Aha moment".

```
08-31-16
--------

Had an interesting thought in the shower. I was thinking about
intermediate testing of the capacitor-based IVS2 and was thinking that I
could just hook the Arduino directly to the laptop for some of the
testing. I'd be able to take measurements and generate graphs on the
Mac. Then it hit me - maybe that's a valid "product" by itself! In fact,
maybe that's what IVS2 should be, period. No Raspberry Pi at all. The
students all have laptops. Not only would it be really cheap, but now we
have our "graphical display". Don't even need an LCD display. What -do-
we need?

 - Arduino
 - Capacitors
 - Shunt resistor
 - Relay or FET
 - ADC
 - Op amps
 - Resistors (voltage divider, op amp, bleed, others?)
 - MC-4 connectors and wires
 - Small box
 - Status LEDs?

We don't even need a battery since the Arduino is powered from the
laptop via USB, and the ADC can run be powered by the Arduino. I don't
think I need any switches or buttons, assuming the Arduino controls the
relay/FET, and the "start" is initiated from the user on the laptop and
sent to the Arduino via USB. One "downside" is that I would have to
basically start over on the software. I'd have to learn how to write a
user-friendly app that would run on Mac or Windows. I'd have to borrow a
windows laptop (Paul?). Of course that is only a downside if I don't
value the learning experience.
```

Figure 1-2: Aha moment

That was really when IV Swinger 2 was born.

Most of the "design objectives" in Section 1.2 were now achievable. But the truth is that they weren't even objectives until after I had reached this point.

1.3 Where Did the Name Come From?

The name "IV Swinger" comes from the expression "swinging out an IV curve", which is how Gil Masters refers to the process of plotting an IV curve using the manual method (light bulb load bank, ammeter, voltmeter). Other people talk about "tracing" or "sweeping out" an IV curve, but I believe Gil is unique in his use of the "swinging" terminology. It is such a reflection of his enthusiasm and positive attitude! It sounds so fast and fun! The reality is that doing it manually is slow and labor intensive - more like "slogging out an IV curve" if you ask me. So I named this device the IV Swinger in the hope that it would make tracing an IV curve as fast and fun as Gil made doing it manually sounded.

2 Overview

2.1 Manual Generation of an IV Curve

It is useful to understand how an IV curve can be generated "by hand" before discussing the high-level design of IV Swinger 1 and its evolution to IV Swinger 2. In addition to meters to measure the voltage and current, a variable resistance load is required. A potentiometer or rheostat can be used for this for small PV modules, but a very large one would be required to be able to withstand the power of a modern PV module. In Gil Masters' class at Stanford, he used a bank of lightbulbs as shown in Figure 2-1.

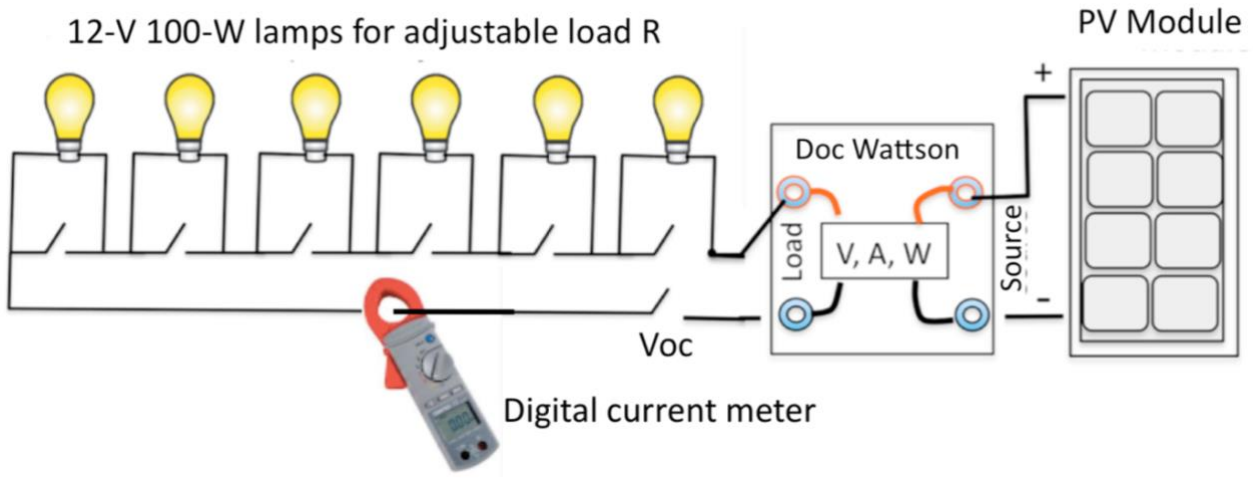


Figure 2-1: Manual IV Curve Tracer (CEE176B lab)

With all of the switches in the closed position, the short-circuit current (I_{SC}) can be measured. With the V_{OC} switch open, the V_{OC} voltage can be measured. With the V_{OC} switch closed and one or more of the lightbulb switches open, the current and voltage at different points on the curve can be measured. All of these manual measurements are then entered into a spreadsheet, and the IV curve is plotted.

2.2 High-level Description of IV Swinger 1

The diagram in Figure 2-2 below represents the IV Swinger 1 at a high level. The design is a very direct automation of the manual process. The bubbles show the analogs to the manual process.

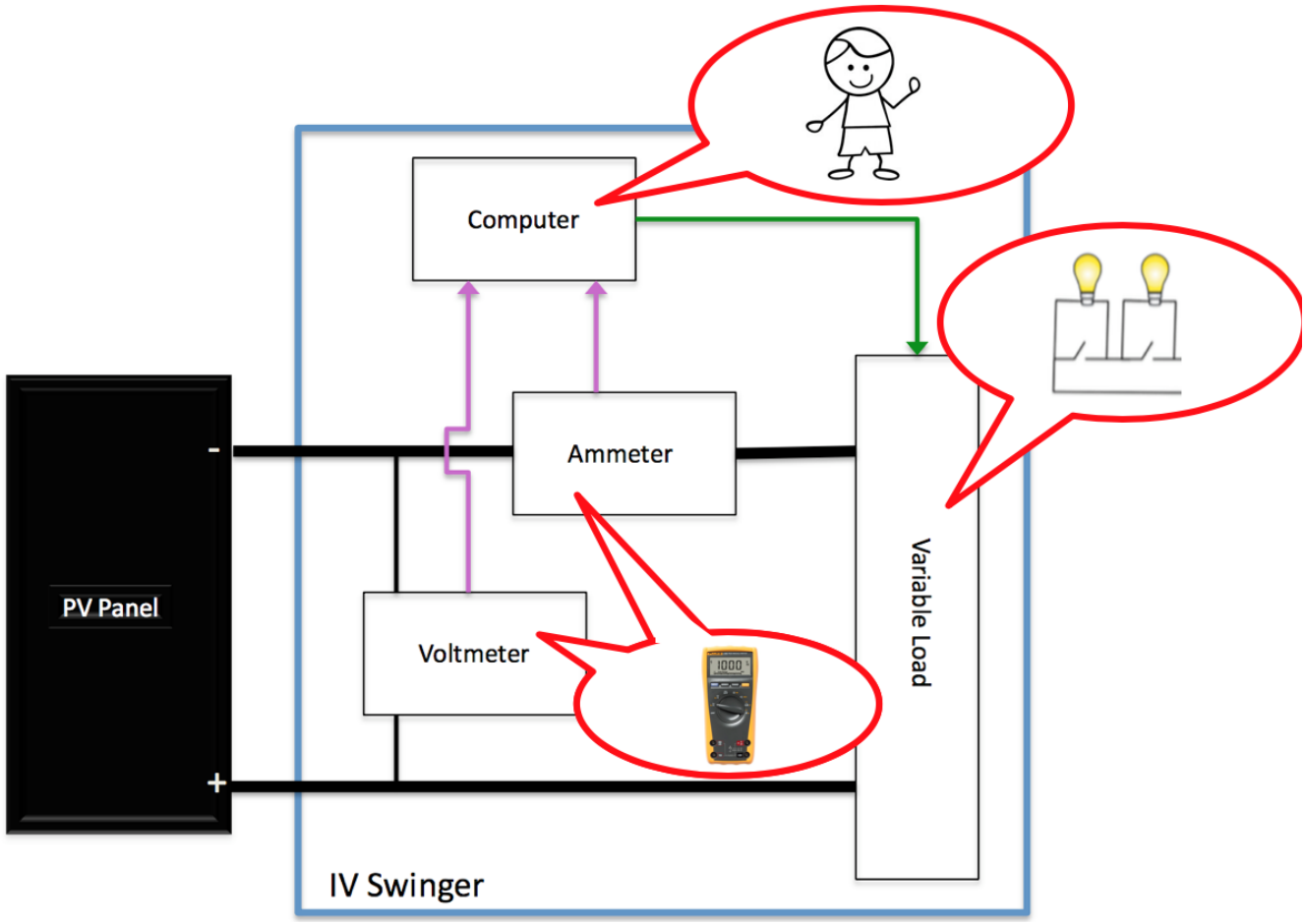


Figure 2-2: High-level Block Diagram of IV Swinger 1

The variable load maps to the light bulb bank and its switches (or potentiometer/rheostat and its knob). The ammeter and voltmeter map to the multimeter. The computer maps to the human.

The bold lines from the PV panel to the variable load represent the load circuit. These are the wires that carry the current generated by the PV panel to and from the load. The ammeter is in series on one leg of the load circuit, measuring the current. The voltmeter is in parallel, between the outputs of the PV panel, measuring the voltage. The computer controls the resistance of the variable load (green arrow) and reads the values from the ammeter and voltmeter (pink arrows).

The variable load is implemented with a chain of immersion heating coils and power resistors. Relays are used to either include or exclude (bypass) each of the loads in the chain. Software running on the computer controls the relays to incrementally increase the resistance of the variable load. At each increment it reads the current and voltage values and records them. The resulting set of data points are used to plot the IV curve.

2.3 High-level Description of IV Swinger 2

Figure 2-2 in the previous section applies to IV Swinger 2 too. However, where IVS1 uses a Raspberry Pi for the computer, IVS2 uses a combination of an Arduino and a laptop. And where IVS1 uses a chain of resistive loads controlled by relays for the variable load, IVS2 uses a capacitor.

2.3.1 The IV Swinger 2 Load

If you understand how the chain of lightbulbs with bypass switches in Figure 2-1 above can be used to generate an IV curve for a PV module, then you can easily understand how the chain of immersion heating coils, power resistors, and relays are used in the IV Swinger 1 design. The IV Swinger 2 load is very simple, but a bit less intuitive to understand.

As shown in Figure 2-3, the IVS2 load consists of three parts:

- A capacitor
- A single-pole double-throw (SPDT) relay
- A bleed resistor

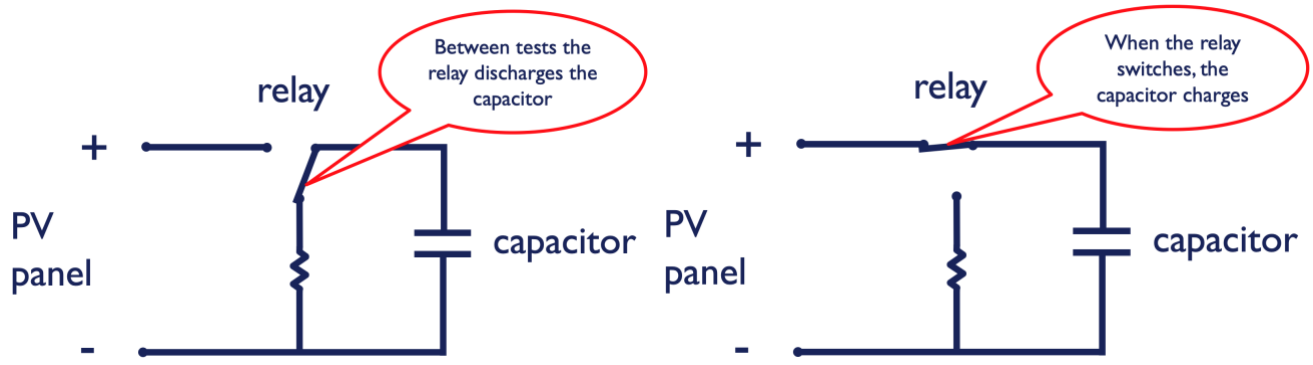


Figure 2-3: IV Swinger 2 Load

The only control input is one signal to activate or deactivate the relay. Unlike the resistive loads of the manual method and IVS1, there is no incremental control of the load resistance. The relay simply determines whether the capacitor charges or discharges (bleeds). In IVS1, the green line in Figure 2-2 tells the variable load, "set the resistance to X ohms". In IVS2, that green line is just a start/stop signal.

A capacitor is not a resistor. However, it has the following properties:

- A discharged capacitor "looks" like a short circuit ($R = 0\ \Omega$)
- A fully charged capacitor "looks" like an open circuit ($R = \infty\ \Omega$)
- A charging capacitor "looks" like all the resistances between ($R = 0\ \Omega \rightarrow \infty\ \Omega$)

This is just what is needed for the variable load: something that varies between a short circuit and an open circuit. The "catch" is that all this happens very quickly once the relay switches. Unlike with the discrete resistive loads of the IVS1 design, the rate that the curve is traced is not under control of software; it is controlled by the physics of the capacitor.

Initially, the flow of positive charges into the capacitor and negative charges out of it is virtually unimpeded, and it appears as if current is flowing through the capacitor, just like a wire, at the maximum current that the PV module can produce (I_{SC}). The voltage across the capacitor is zero. But as soon as the first electrons start flowing, the capacitor starts to charge, and the flow starts seeing some resistance. The voltage across the capacitor starts rising. When the voltage across the capacitor reaches the maximum voltage that the PV panel can produce (V_{OC}), an equilibrium is reached, current stops flowing, and the capacitor is fully charged. The capacitance of the capacitor determines how much time it takes to

go from discharged to fully charged. This time is a fraction of a second for capacitors that are a reasonable physical size and cost.

2.3.2 The IV Swinger 2 Computer(s)

The Raspberry Pi computer that is used for IVS1 would be fast enough to be able to "keep up" with the speed that the capacitor charges up. However, the normal Linux operating system that runs on Raspberry Pi is not a real-time OS, meaning it is difficult to guarantee that it won't switch to a different task right in the middle of the critical time that we need it to be reading the ammeter and voltmeter values.

A much better solution is to use an Arduino microcontroller. The Arduino is less expensive than a Raspberry Pi and is smaller. The software that runs on the Arduino does not run under the control of an OS, so there is no way it can be interrupted when it is performing the time-critical task of reading the ammeter and voltmeter values while the capacitor is charging up.

The downside of the Arduino is that it cannot be used for all of the other required computing tasks such as actually producing a graph of the IV curve. While it would be possible to use a Raspberry Pi for these tasks, that would add to the cost (and size) of IVS2. Instead, a general-purpose Mac or Windows laptop is used. Since it can be assumed that the user already owns such a device, its cost does not have to be accounted for. More importantly, a laptop is more powerful than a Raspberry Pi and it has a high-resolution color display. Furthermore, the Arduino and other electronics can be powered from the laptop USB port that is also used for communication.

2.3.3 The IV Swinger 2 Ammeter and Voltmeter

The ammeter and voltmeter circuits for IVS2 are very similar to the ones used for IVS1. The main difference is that a different analog-to-digital converter (ADC) was needed that could both take measurements and communicate their values more quickly. Details will be discussed later in the document.

2.3.4 IV Swinger 2 Variants

Several variants of IV Swinger 2 have been developed, and all will be discussed in this document:

- **PV module version using a** Perma-Proto circuit board **and** electromagnetic relay (EMR)
- PV module version using a printed circuit board (PCB) and electromagnetic relay (EMR)
- PV cell version using a printed circuit board (PCB) and electromagnetic relays (EMRs)
- PV module version using a printed circuit board (PCB) and solid-state relays (SSRs)
- PV cell version using a printed circuit board (PCB) and solid-state relays (SSRs)

The Perma-Proto versions have exactly the same circuit designs as their PCB equivalents. The SSR versions do not have Perma-Proto equivalents.

The baseline variant is the EMR-based PV module version. This document describes that variant in detail. The other variants are described only in terms of their differences from the baseline in Chapter 7 starting on page 63.

2.4 Baseline IV Swinger 2 Schematic

Figure 2-4 below is a schematic of the baseline IV Swinger 2 design. It includes everything that is “in the box”, including the Arduino and the relay module. It does not show the laptop, which connects to the Arduino with a USB cable.

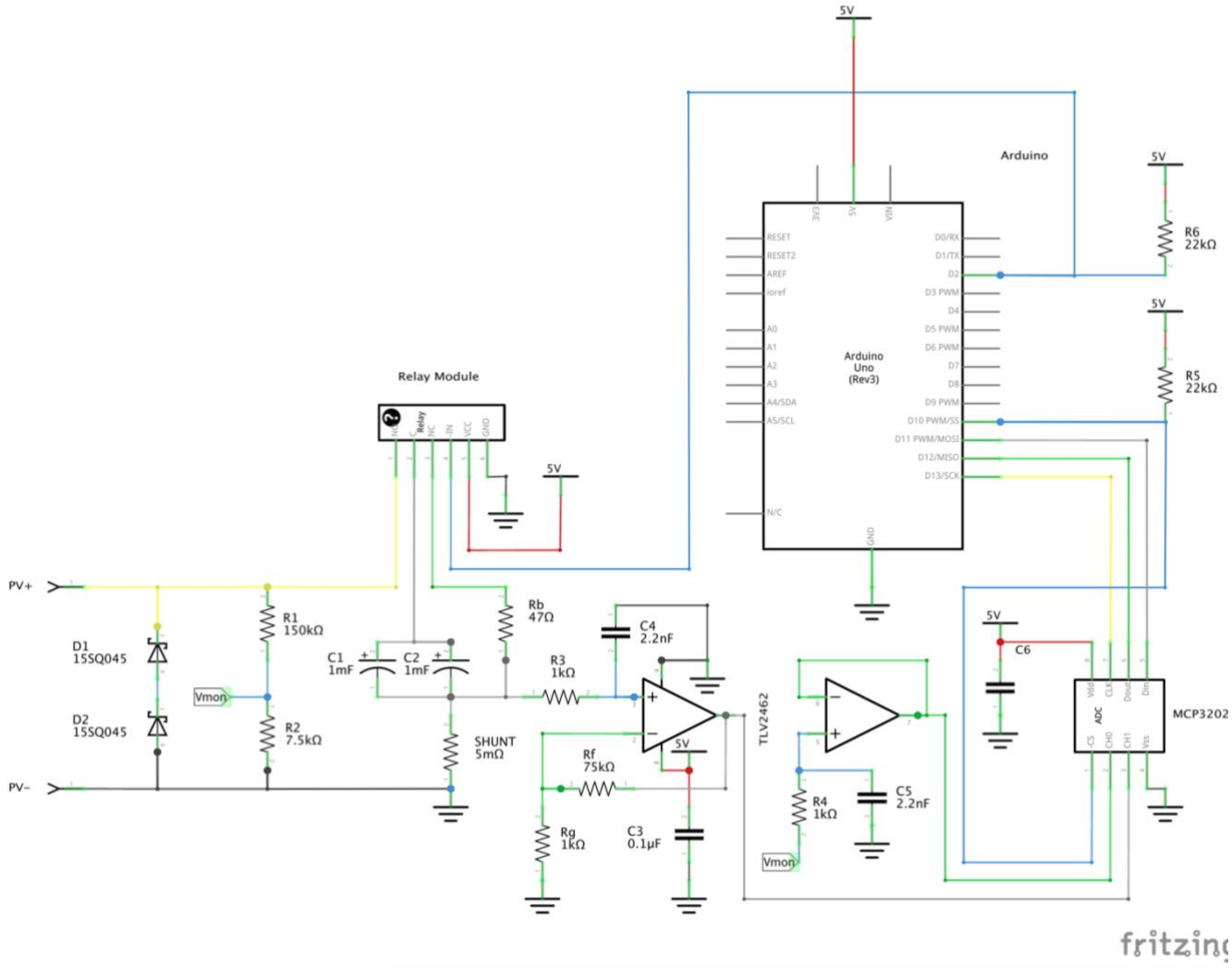


Figure 2-4: Baseline IV Swinger 2 Schematic

3 Load Circuit Design

The load circuit carries the current generated by the PV module. It starts at the red binding post (PV+) and ends at the black binding post (PV-). Its components are: the binding posts, the bypass diodes, the relay module, the load capacitors, and the bleed resistor. Figure 3-1 below shows where these components are on the schematic. Each of these components will be discussed in a section in this chapter. Note that the shunt resistor and resistors R1 and R2 are part of the ammeter and voltmeter and will be discussed in the next chapter.

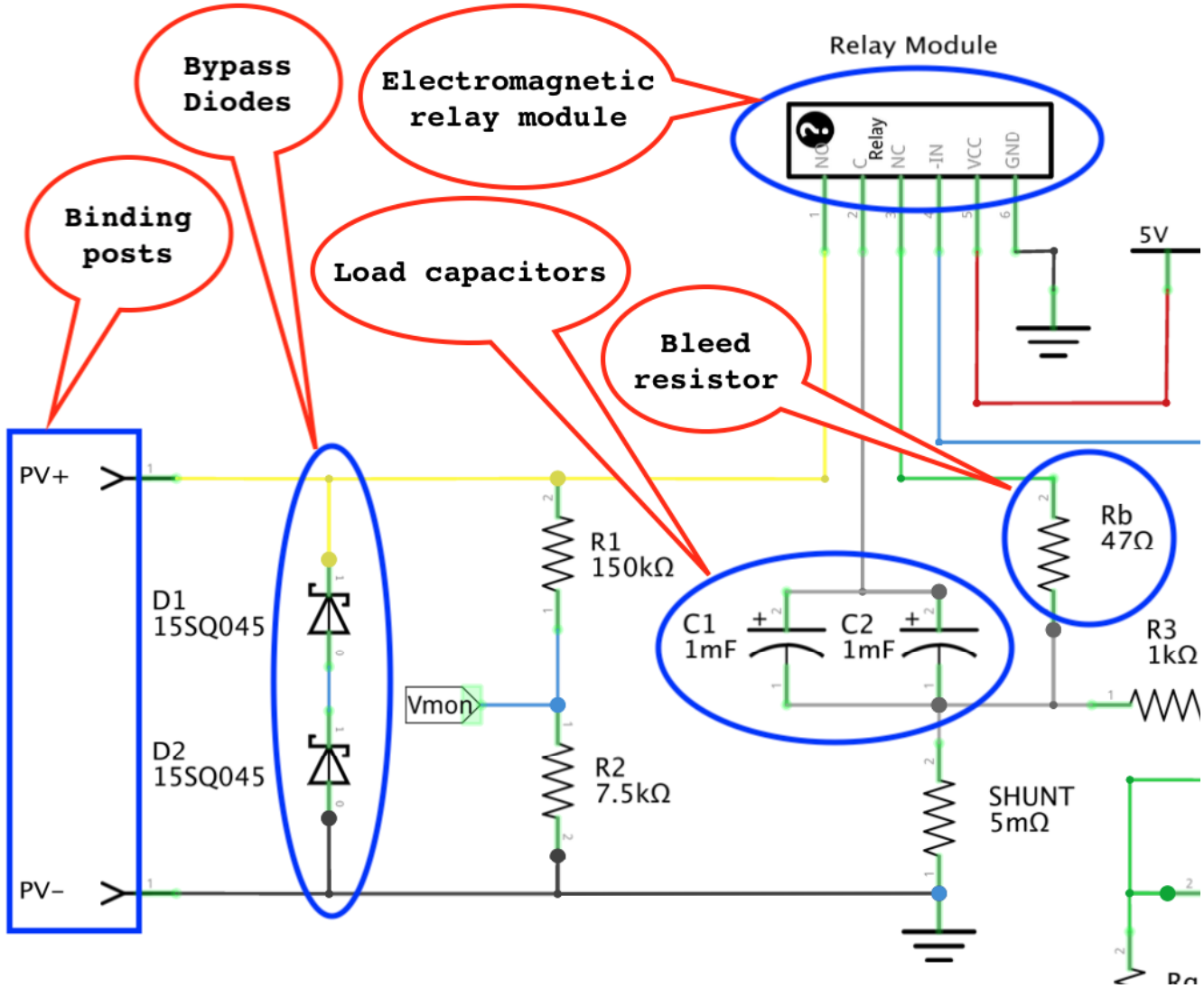


Figure 3-1: Load Circuit Components

Figure 3-2 shows the current flow path when the relay is active (ON) and Figure 3-3 shows the current flow path when the relay is inactive (OFF).

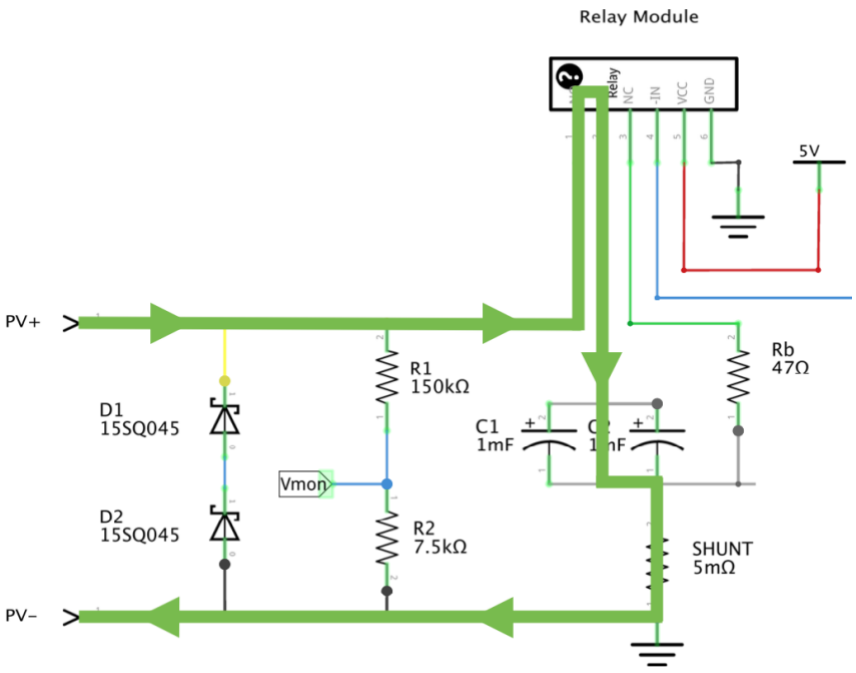


Figure 3-2: Load Circuit "ON"

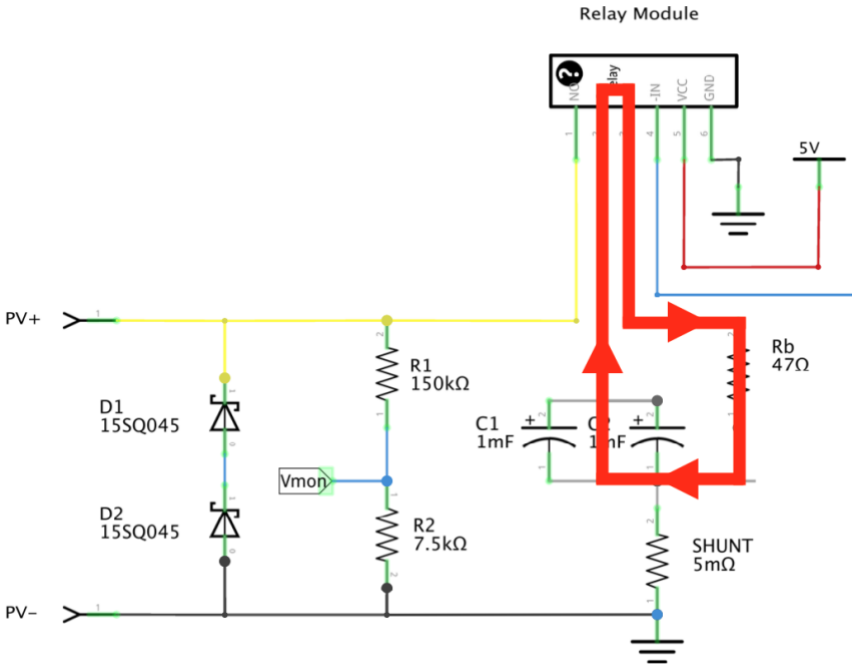


Figure 3-3: Load Circuit “OFF”

3.1 Binding Posts

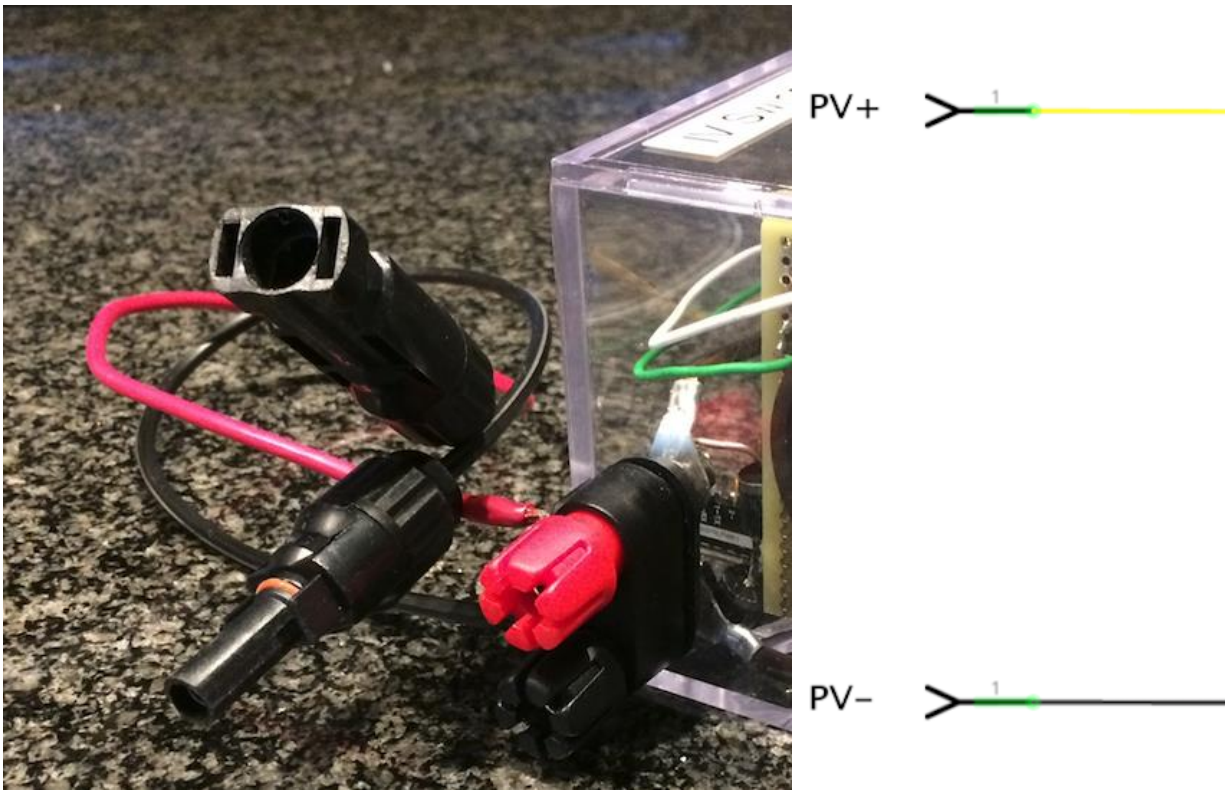


Figure 3-4: Binding Posts with PV Cables

The connections to the PV module are made via binding posts. The red binding post is connected to the PV+ input and the black binding post is connected to the PV- input. The binding posts provide connectivity flexibility. Typically, they will be connected to wires with MC-4 connectors as shown in Figure 3-4 above. In this case the wires are simply stripped and inserted into the holes on the sides of the binding posts and the knobs tightened down to bind the wires in place. The binding posts also accept banana plugs, which are easier to connect and disconnect. With the IVS2 lid off, the threaded posts on the inside are convenient for connecting alligator clips. All of these are very nice, not only for connecting a PV module, but also for connecting a multimeter or bench power supply (used for calibration).

3.2 Bypass Diode(s)

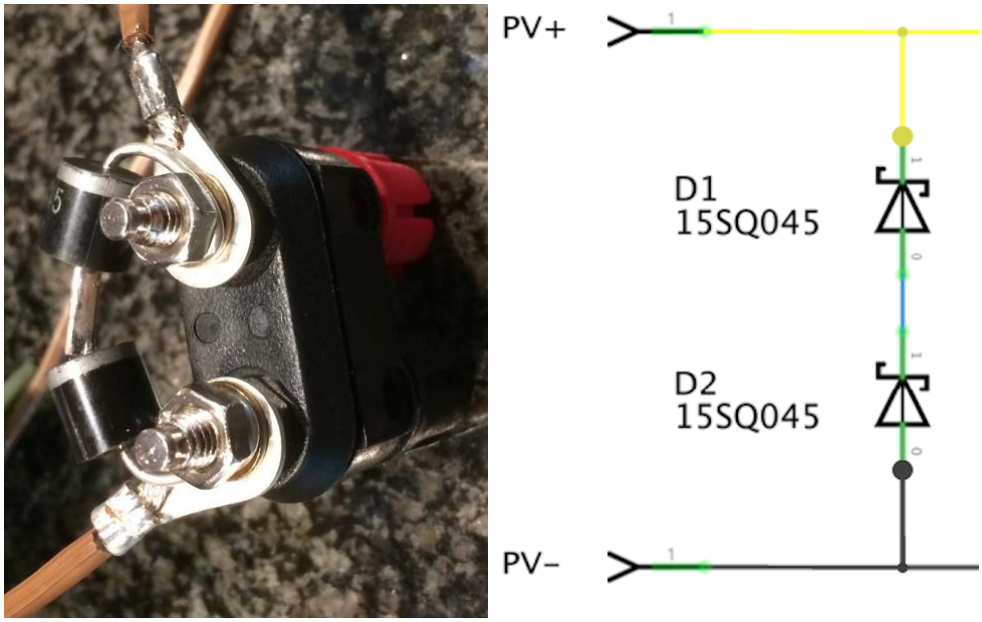


Figure 3-5: Bypass Diodes

Diodes D1 and D2 are Schottky diodes of the type used internally in PV modules to bypass current around a shaded segment of the module. Their purpose in the IVS2 design is to protect the internal components (especially the load capacitors) against inadvertent reverse connection of the PV module. If such a connection is made, the diodes will conduct, and the negative input voltage will be reduced to a small value.

The 15SQ045 diodes are rated at 15 A and 45 V. This means that they can safely pass 15 A of current continuously. Since IVS2 is rated for PV modules that have an I_{SC} of up to 10 A, that is sufficient to protect the internals even if the PV module is connected backwards indefinitely (in full sun).

The 45 V rating refers to the highest voltage that can be applied in the reverse-biased direction before the diode will break down and conduct in that direction (and once this happens, the diode is ruined and will conduct in the reverse direction even at lower voltages). When the PV module is connected correctly, the diode pair is reverse-biased at the V_{OC} of the module. Therefore, a single diode would be adequate for PV modules up to V_{OC} = 45 V. The intent of putting two in series was that breakdown voltages would add, and would be adequate for a 90 volt V_{OC}. This is **NOT CORRECT**, however, as described in an answer on Stack Exchange. **This means that two 45 V diodes are really no better than one and will not handle PV modules with a V_{OC} greater that 45 V.**

The original reason for using 45 V diodes was that they are inexpensive and available on Amazon. But 100 V diodes (15SQ100) are available from DigiKey for less than $1. A single 15SQ100 diode is now recommended in place of the two 15SQ045 diodes.

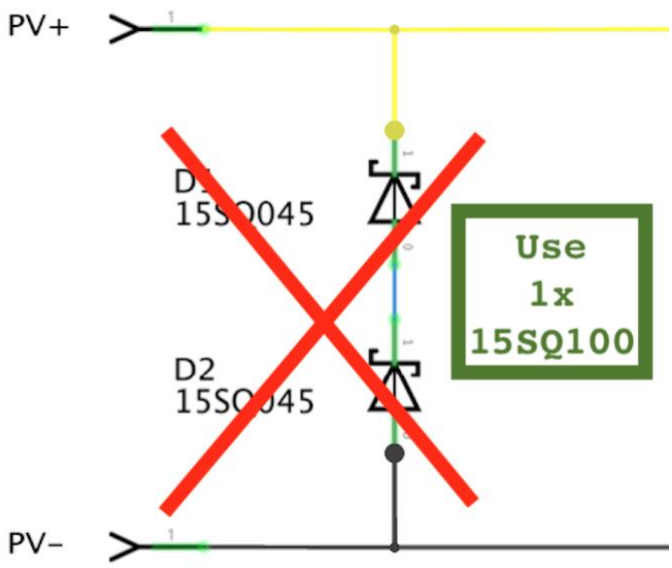
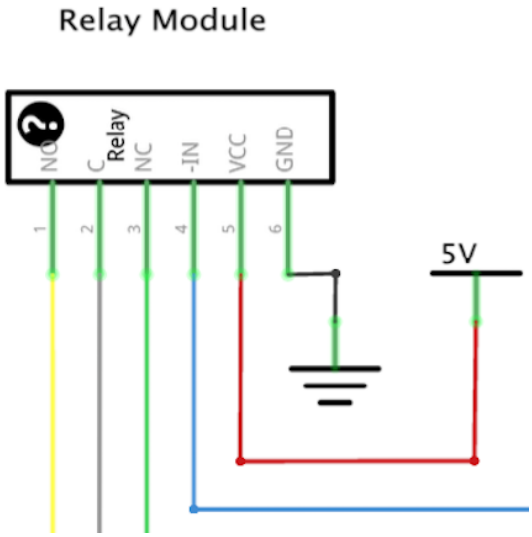


Figure 3-6: Use Single 100V Bypass Diode

3.3 Electromagnetic Relay (EMR) Module



A single-pole double-throw (SPDT) electromagnetic relay (EMR) is a device that uses an electromagnet to switch a common (C) terminal from being connected to a “normally closed” (NC) terminal to being connected to a “normally open” (NO) terminal as shown in Figure 3-7 below.

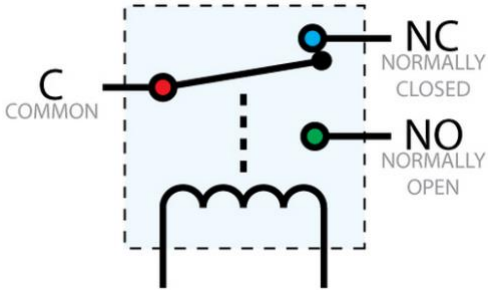


Figure 3-7: SPDT relay schematic drawing

Physically, the inside of a relay is shown in Figure 3-8 below.

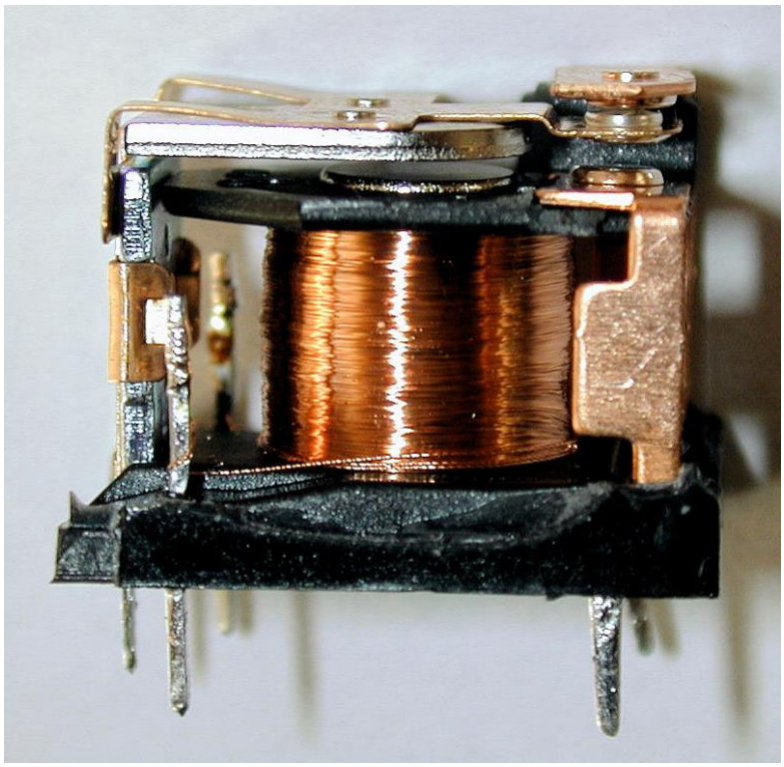


Figure 3-8: Inside a physical relay

When there is no current flowing through the coil, the electromagnet is “off” and the spring metal holds the middle (C) contact up against the NC contact. When current flows through the coil, the electromagnet is “on”, and it pulls the C contact down to the NO contact. When a relay switches there is a very audible “click”.

When the relay is not activated, the C terminal is connected to the NC terminal and the bleed resistor is connected across the load capacitors as in Figure 3-3 on page 26. When the relay is activated, the C terminal is connected to the NO terminal and the PV+ input is connected to the load capacitors as in Figure 3-2 on page 26.

Relay modules are readily available and very inexpensive. In addition to the relay itself, the modules have other necessary supporting components all mounted on a PCB that allows for easy mounting to an enclosure. Figure 3-9 below is a photograph of the EMR module of the type used in IVS2 along with a circuit diagram (this module does not have the jumper that is shown in the diagram).

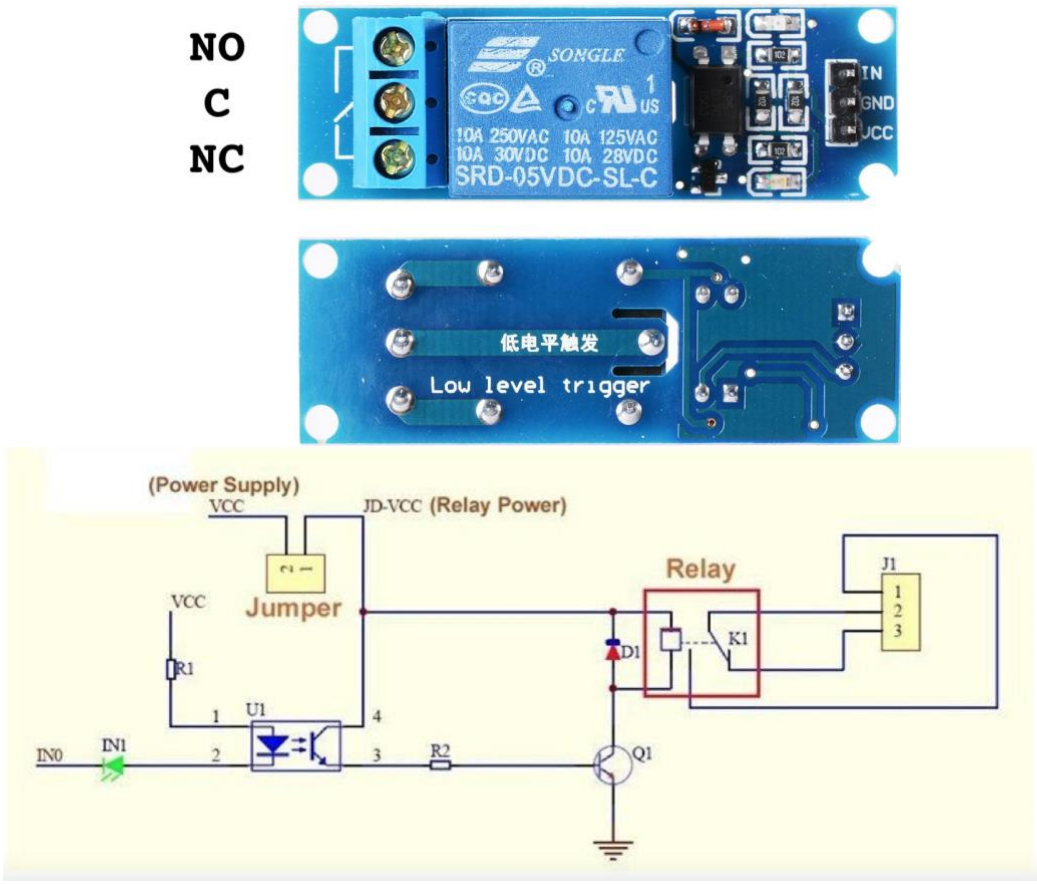


Figure 3-9: EMR module

The relay itself is the blue box in the top photo. To the left of the relay is the screw terminal block where the connections are made. The middle connection is the common (C) terminal. The one on the top is the Normally Open (NO) terminal, and the one on the bottom is the Normally Closed (NC) terminal. Note the little drawing on the silkscreen that helps to remember which terminal is which, even if you can't remember their names. The terminal block has holes where the wires are inserted and tiny screws to hold them in place.

The green LED that lights up when the relay is active is near the bottom, on the right end of the board. There is also a red LED at the top edge that lights up when power is applied, but this is not shown on the circuit diagram. The other components on the board are an optoisolator, a transistor, resistors and a diode, which are all shown in the circuit diagram.

The 3 pins on the right end of the board are -IN, GND and VCC. In IVS2, the GND pin is connected to the common ground used by all components (also tied to the negative side of the PV). VCC is connected to +5V from the Arduino. The -IN pin is connected to Arduino pin D2 to control whether the relay is activated or deactivated (see Figure 3-10 below). This pin is "active low", which means that a low (near GND) voltage activates the relay and a high (near +5V) voltage deactivates the relay. The words "Low level trigger" on the back of the module indicate this. R6 is a 22kΩ pull-up resistor to hold the relay in the inactive state before the Arduino software defines pin D2 as an output and starts driving it high (inactive).

Some EMR modules are designed such that the input is active-high, and some have a jumper to allow them to work either way. The software is written to work with active-low input EMR modules. However, there is configuration option in the software to allow it to work with an active-high EMR module. When an active-high EMR module is used, resistor R6 does the opposite of what it was intended to do; it activates the relay state before the Arduino software defines pin D2 as an output and starts driving it low (inactive). The fact that this works implies that resistor R6 is not necessary.

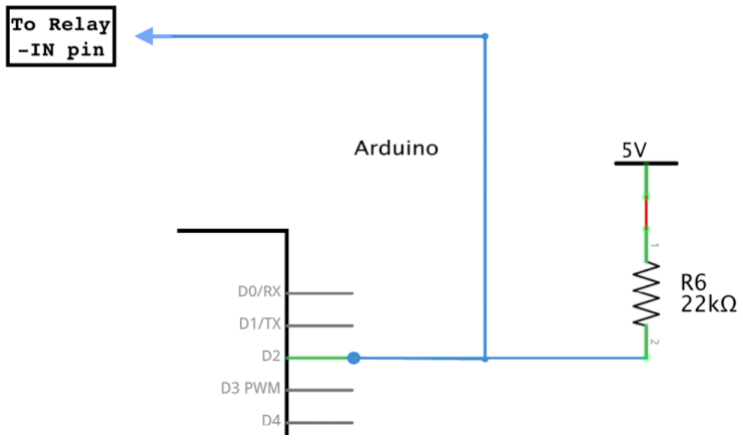


Figure 3-10: Relay module -IN pin control from Arduino

3.3.1 Cost

The recommended EMR module is sold in a pack of 5 on Amazon for $8.99, so the unit cost is about $2.

3.3.2 Current and Voltage Limitations

The important specifications are those of the Songle SRD-05VDC-SL-C relay. The most important of those is how much load current it can handle. The case has some values printed on it:

It's not clear what the difference is between the ones on the left and the ones on the right, but the current is 10A in both cases. The modules are always advertised as handling up to 10A. Since 10A is the maximum I_{SC} we have chosen, this sounds good.

The actual Songle relay specification has the following table, however:

7. CONTACT RATING

Item \ Type	SRD	
	FORM C	FORM A
Contact Capacity Resistive Load (cosΦ=1)	7A 28VDC 10A 125VAC 7A 240VAC	10A 28VDC 10A 240VAC
Inductive Load (cosΦ=0.4 L/R=7msec)	3A 120VAC 3A 28VDC	5A 120VAC 5A 28VDC
Max. Allowable Voltage	250VAC/110VDC	250VAC/110VDC
Max. Allowable Power Force	800VAC/240W	1200VA/300W
Contact Material	AgCdO	AgCdO

The relays used on the module are "Form C" (as indicated by the C at the end of the part number). So it appears that the maximum current is 7A for a resistive load and only 3A for an inductive load, and the markings on the case are misleading. This was a larger concern for IVS1, because the I_{SC} current flowed for a much longer time in that design. In the IVS2 design, the I_{SC} current is flowing for an extremely brief amount of time, so it is not an issue.

As for voltage, it is unclear what the significance is of the 28V value. The table says that the maximum allowable voltage is 110 VDC, which is higher than our maximum V_{OC} value of 80V.

It is still likely that the EMR is the most failure-prone component in IVS2. But it is very easy and inexpensive to replace if it does wear out. The SSR-based designs eliminate this weak point.

3.3.3 Switching Conditions

Despite the fact that the voltage and current ratings are marginally adequate (or, arguably, inadequate) for the specified limits of the IVS2, the conditions that are present when the relay switches are a mitigating factor.

Relays wear out and fail when they are switched while current is flowing. This is because the current cannot stop instantly due to the inductance of the circuit. All circuits have some amount of inductance. The result is arcing, which eventually destroys the relay contacts[2].

The IVS2 relay is never switched when current is flowing. Between tests, it is in the inactive state; the load circuit is open so no current is flowing when it is switched from the inactive to the active state. It is not switched from active back to inactive until the capacitor has charged up, at which point the current is once again nearly zero.

[2] Section 3.2.6 of the "IV Swinger: Design, Construction, and Operation" document has a thorough explanation of arcing, if you are interested in knowing more.

In both switching cases, current does begin to flow when the relay is in its new state. One might think that it would arc during this transition, but that is not the case. This is because the circuit inductance works in our favor on this transition; current cannot instantly change from zero to a high value. At the beginning of the IV curve, there is actually a ramp up to the I_{SC} value that the Arduino software filters out. This is due to the capacitors' parasitic inductance. When the relay switches back to the bleed resistor, the current also ramps up to its maximum bleed rate. This ramp is due to the parasitic inductance of the bleed resistor and the capacitors.

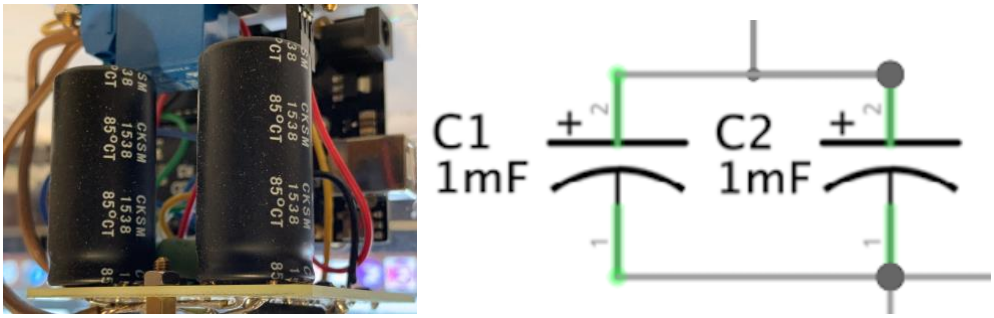
3.3.4 Current/Power Consumption

The EMR requires a negligible amount of power when it is in the inactive state, but it requires about half a watt of power in the active state. This is the power required to energize the electromagnet coil.

- Coil current: 89.3mA
- Coil resistance: 55Ω ±10%
- Coil power: 0.39W – 0.48W

In the IVS1 design there were 16 EMRs and this was a significant factor to take into account. Since the IVS2 design only has one, and it is active for a very brief time, this is almost negligible. However, the current draw of the EMR coil does have an interesting effect. It causes a droop in the +5V supplied from the laptop via USB. This affects the ammeter and voltmeter measurements because they use the +5V as a reference. This will be discussed more in Section 4.2.1.5 on page 47.

3.4 Load Capacitors



Conceptually the load is a single capacitor, but for reasons that will be explained shortly, two physical capacitors, C1 and C2, are used in parallel.

Capacitors are rated primarily by two metrics: capacitance and voltage. One more metric that can be relevant is their equivalent series resistance (ESR). The cost and physical size of a capacitor are generally proportional to their voltage and capacitance, and low ESR costs more. So, in order to keep the cost and size down, we want to find the lowest voltage that is adequate, the lowest capacitance that is adequate, and the highest ESR that is adequate.

3.4.1 Voltage Requirement

The voltage rating of a capacitor specifies the highest voltage that can safely be applied between its two terminals. Higher voltages than that can destroy the capacitor. It is generally accepted that capacitors should be rated around 20% higher than the highest voltage that they are expected to encounter in the

application. In our case, 80 V is the rated maximum PV module V_{OC}, so 100 V capacitors are adequate. Choosing capacitors with a higher voltage rating would be more expensive, and the capacitors would be physically larger.

3.4.2 Capacitance Requirement

The capacitance determines how long it takes the capacitor to charge up. If the capacitance is too low, the Arduino will not be able to take measurements fast enough for the curve to have the desirable resolution.

To choose the ideal capacitance, we need to know:

- How fast can the Arduino take measurements?
- What is the desirable resolution?

Armed with the answers to those two questions, we can use some physics and math to determine the minimum required capacitance.

Initial estimates were that the Arduino would be able to take one pair of (I, V) measurements every 100 microseconds (µs) using the MCP3202 analog-to-digital converter (ADC), which will be discussed in Chapter 4. This included some estimated software processing time. That analysis is not included here, but it turned out to be a bit conservative; the actual time is about 65 µs per point. We'll use 100 µs for the analysis in this chapter since that was the basis on which the actual decisions were made, and it makes the math easier to follow.

The desirable resolution is harder to quantify because more resolution is always better. But it does come at a cost, and there is certainly a point of diminishing returns. In reality, the analysis was done the other way around. In other words, IV curves were modeled using different capacitance values, and then subjective criteria were used to decide what capacitance to use. Those subjective criteria were based not only on how the resolution looked, but also took into account the physical size, cost and availability of actual capacitors. Another factor to consider was how much time it would take to trace the entire curve; added resolution comes at the cost of a longer trace time.

This brings us to the relationship between capacitance and time. The resolution is the distance between points on the IV curve. We need to know how far along the IV curve we will have moved in 100 µs. The answer to this is not a constant; it depends on the specific current and voltage of the two points, i.e. it depends on the characteristics of the curve (I_{SC}, V_{OC}, shading) and which part of the curve is being traced.

Physics helps us here. The current through[3] a capacitor is given by the following equation:

Equation 3-1: Current at one moment in time

$$i = C \bullet \frac{dv}{dt}$$

[3] Current does not actually flow "through" a capacitor; it flows into and it flows out of it. But from the connected circuit's point of view, the current appears to flow through the capacitor.

This says that at any *moment in time*, the current is equal to the capacitance (C) times how fast the voltage is changing (*dv/dt*). The unit of capacitance is the farad (F), so the units on the right-hand side of the equation are F•V/s. One F•V/s is one amp (A). One moment in time is one point on the IV curve.

For any two points on the IV curve that are reasonably close together in time, their average current can be approximated by the following modification of Equation 3-1:

Equation 3-2: Average current between two close points

$$I_{avg} \approx C \bullet \frac{\Delta V}{\Delta t}$$

This says that the average current between two close points is approximately equal to the capacitance times the difference in their voltages divided by the time difference.

We can now solve for Δt, ΔV or C.

Equation 3-3: Solve for Δt

$$\Delta t \approx C \bullet \frac{\Delta V}{I_{avg}}$$

Equation 3-4: Solve for ΔV

$$\Delta V \approx \frac{I_{avg} \bullet \Delta t}{C}$$

Equation 3-5: Solve for C

$$C \approx I_{avg} \bullet \frac{\Delta t}{\Delta V}$$

Let's look at an example. Figure 3-11 below shows a typical IV curve with two pairs of points that we will be looking at.

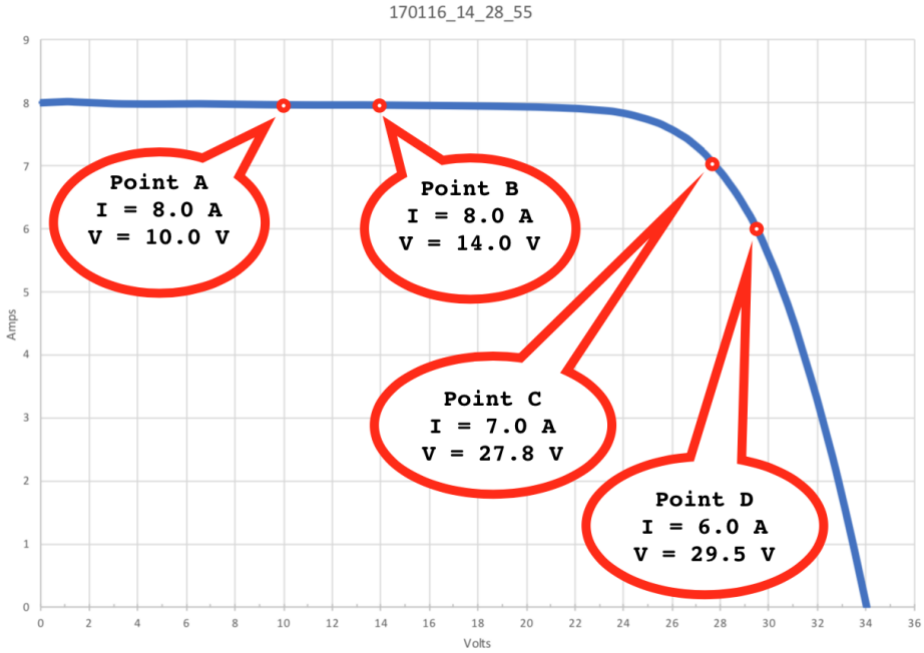


Figure 3-11: Δt example IV curve

3.4.2.1 Solving for Δt

First, let's take the capacitance value that we ended up choosing (2000 µF), and calculate how much time passes (Δt) between the two points in each of these pairs. We'll use Equation 3-3 for this.

Point A to Point B:

- C = 2000 µF = 0.002 F
- ΔV = 14.0V – 10.0V = 4.0V
- I_{avg} = 8.0A

$$\Delta t \approx 0.002\ \text{F} \bullet \frac{4.0\ \text{V}}{8.0\ \text{A}} = 0.001\ \text{s} = 1\ \text{ms} = 1000\ \mu s$$

If the Arduino can measure one point every 100 µs, this means 1000/100 = 10 sub-segments in this segment.

Point C to Point D:

- C = 2000 µF = 0.002 F
- ΔV = 29.5V – 27.8V = 1.7V
- I_{avg} = 6.5A

$$\Delta t \approx 0.002\ \text{F} \bullet \frac{1.7\ \text{V}}{6.5\ \text{A}} = 0.000523\ \text{s} = 523\ \mu s$$

Although the distance on the graph is approximately equal to the distance between points A and B, the time between points C and D is about half. Only 523/100 ≈ 5 sub-segments can be measured.

3.4.2.2 Solving for C

Steep descents at high currents are the hardest to keep up with, i.e. for a fixed sampling rate, the point density will be the lowest at these parts of the curve.

Knowing that fact, the required capacitance could be determined using Equation 3-5 if we can decide how many points we would like to see in such a segment. For example, let us assume that we would like to see at least three points between points C and D (i.e. four sub-segments).

- Δt = total time between C and D = 4 • 100 µs = 400 µs = 0.0004 s
- ΔV = 29.5V – 27.8V = 1.7V
- I_{avg} = 6.5A

$$C \approx 6.5\ A \bullet \frac{0.0004\ s}{1.7V} \approx 0.0015\ \text{F} \approx 1500\ \mu F$$

3.4.2.3 Resolving inflections

In Section 3.4.2.2 above, it was concluded that a capacitance of 1500 µF would be adequate to resolve the C-to-D segment into four sub-segments. Is that enough? Perhaps it is for this example curve. This isn't the worst case, however. The I_{SC} is 8 A in this example, but IVS2 supports I_{SC} values up to 10 A. The analogous segment for an IV curve with an I_{SC} of 10 A will have an average current of about 8.5 A, which would require a 1960 µF capacitance. A PV module with a lower internal resistance and/or shorter cables will have an IV curve with a steeper descent (i.e. smaller ΔV), so that would dictate a higher capacitance.

Furthermore, the real motivation for higher resolution is not curves like the example in Figure 3-11. Even with only ~20 points per curve, IVS1 was able to produce very nice curves for full-sun cases by using Catmull-Rom spline interpolation. Where it fell short was shading cases for PV modules with bypass diodes. Figure 3-12 below shows a shading case plotted with IVS1. The inflection point after the first descent caused the interpolation to go bad and there is overshoot. Some cases were much worse than this one.

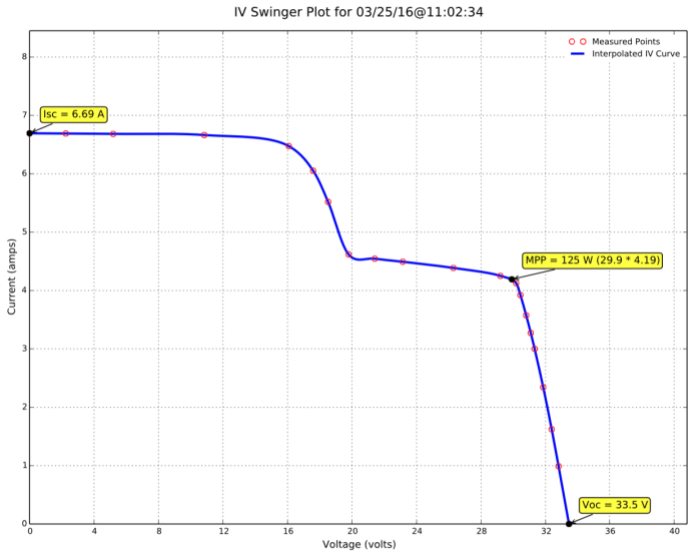


Figure 3-12: Shading inflections with sparse points (IVS1)

Figure 3-13 below shows a very similar case with the much higher resolution of IVS2. The inflection point after the first descent is nice and sharp, which reflects reality.

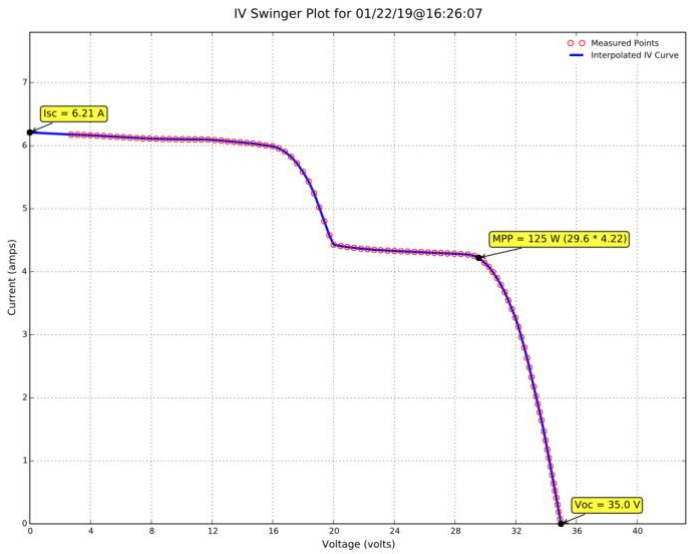


Figure 3-13: Shading inflections with dense points (IVS2)

The most difficult cases for the capacitive load are when there is a much smaller amount of shading, and inflection points are at a much higher current such as in Figure 3-14 below.

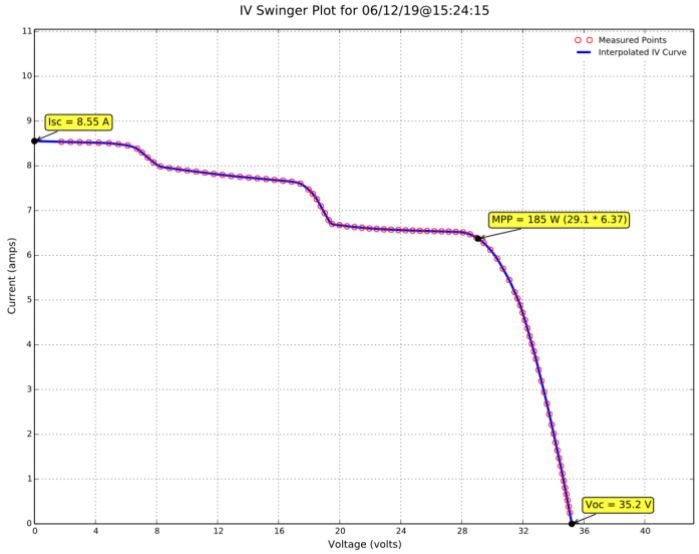


Figure 3-14: Minor shading (inflections at high current)

Of course, Figure 3-13 and Figure 3-14 were generated with the actual IVS2 long after the decision had been made on what capacitance to use. But during the design phase, it was known that such cases should be handled well, and it was not difficult to approximate what they would look like at the inflection points using Equation 3-3 with capacitance values of available capacitors that met the other criteria.

3.4.3 Equivalent Series Resistance (ESR) Requirement

An ideal capacitor has zero resistance when it is completely discharged. A real capacitor can be modeled as an ideal capacitor in series with a small resistor, i.e. even when it is completely discharged, there is a small resistance. This is called the equivalent series resistance (ESR).

Ideally, we would like to have as low an ESR as possible. This is because we would like to be able to measure a point as close to the I_{SC} point as possible so extrapolation to the I_{SC} will be accurate. The true I_{SC} point (V=0) could only be measured if the load could truly be set to zero resistance. Wire resistance, the resistance of the relay contacts, the shunt resistor, and the capacitor ESR all add up to some value that determines the lowest voltage that can actually be measured for a given I_{SC}.

Capacitor specifications often (but not always) specify an ESR value. Confusingly, the ESR value is always specified at a given AC frequency. Contrary to a lot of incorrect information on the web, the actual ESR *does not depend on frequency*. But the most common methods of measuring it are AC tests that attempt to factor out the capacitive reactance to infer the ESR. Since capacitive reactance is inversely proportional to frequency, one method is simply to use a high enough frequency that the capacitive reactance is negligible compared to the ESR. The bottom line is that a capacitor's specified ESR should be close to what we're interested in.

When two capacitors are used in parallel, their capacitances add. If their ESRs are the same, the effective ESR of the two parallel capacitors is half of the individual ESR. This isn't a sneaky way to reduce ESR, though, because a single capacitor with double the capacitance generally has half the ESR.

The ESR for 100 V, 1000 µF capacitors ranges from under 100 mΩ to over 300 mΩ (50 mΩ to 150 mΩ for two in parallel). The worst-case resistance of the rest of the load circuit is about 120 mΩ, but is typically around 30 mΩ. Therefore, the worst-case total is around 270 mΩ. At 10 A, that equates to a voltage of 2.7 V. More typically, the first measured point has a total load circuit resistance of about 120 mΩ, which equates to 1.2 V at the first measured point at 10 A. It turns out that due to relay contact "bounce" and other factors, the first few points often are discarded and the first non-discarded point may have a voltage of 4 V or higher.

For a 60-cell module with a V_{OC} on the order of 36 V with three bypass diodes, the IV curve of a 2/3 shaded case has its MPP around 10 V. If the first measured point is at 4 V, the I_{SC} point can still be extrapolated correctly. It is a non-issue for non-shaded cases.

In conclusion, the difference between the highest ESR capacitors and the lowest ESR capacitors will equate to about a 1 V difference in the voltage of the first measured point when the I_{SC} is 10 A. It will be a smaller difference for lower I_{SC} cases. This will not substantially affect the quality of results.

3.4.4 Physical Size

The physical size (volume) of a capacitor is roughly proportional to the energy that it can store.

Equation 3-6: Capacitor energy storage

$$Energy = \frac{1}{2}CV^2$$

For a given capacitance, a higher voltage capacitor will be larger than a lower voltage capacitor. For a given voltage, a higher capacitance capacitor will be larger than a lower capacitance capacitor.

3.4.5 Type

There are many types of capacitor that have different uses and characteristics. Figure 3-15 below shows the voltage and capacitance ranges for various types. The whole area from 10V to 200V with capacitances from 1mF (1000µF) to 1F is covered by the aluminum electrolytic type only, so there really is no other choice. Fortunately, these are very common and inexpensive.

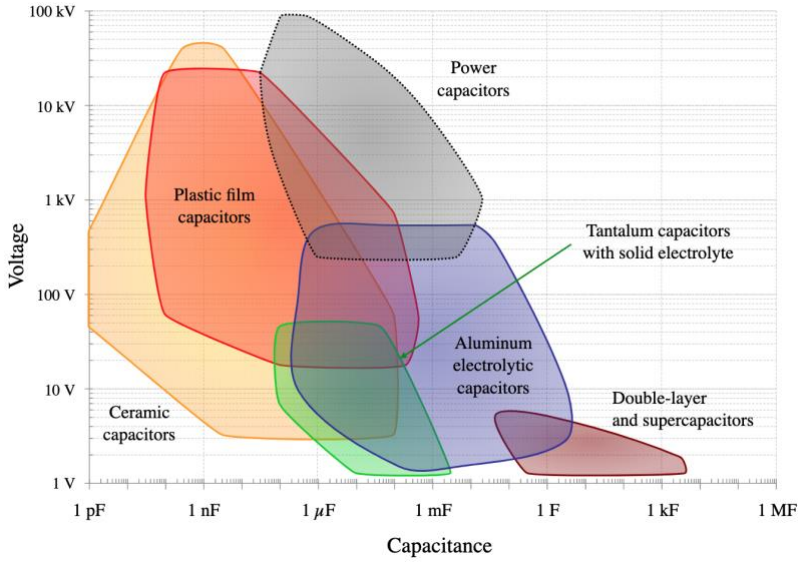


Figure 3-15: Capacitor Types[4]

An important thing to understand is that electrolytic capacitors are polarized. This means that they will operate correctly only if the voltage is higher on the anode (+) terminal than on the cathode (-) terminal. If they are connected backwards, they can fail (even explode).

3.4.6 Form Factor

Electrolytic capacitors are cylindrical and are available in radial and axial packages. Axial capacitors have one lead coming out of each end and are designed to lie flat on the circuit board. Radial capacitors have both leads on the same end and are designed to stand up on the board. Based on circuit board space versus enclosure space tradeoffs, radial was the clear choice.

3.4.7 Cost

The cost of an aluminum electrolytic capacitor is generally proportional to its size. Another factor is popularity; economies of scale bring down the cost of capacitors that are produced in huge numbers, so it is preferable not to choose an "oddball" capacitor just because it seems to hit a perfect sweet spot.

[4] https://commons.wikimedia.org/w/index.php?curid=20463873

3.4.8 Final Choice

Voltage: 100V
Capacitance: 2000 µF (2 x 1000 µF)
ESR: not critical
Package/Case: radial/can
Lead spacing: 7.5 mm (through hole)
Diameter: 18 mm
Height: 33.5 mm – 42 mm

A total capacitance of 2000 µF was chosen because modeling of inflections such as those in Figure 3-14 appeared to be quite good. However, 100V/2000µF is not a common size. 100V/1000µF is very common, however. Two 1000 µF capacitors in parallel are equivalent to one 2000 µF capacitor, so that is what was chosen.

There are many choices of capacitors that meet the requirements, and they are all very similar. Height is one variable. There is a range of prices, but all are under $4 each. The part that is currently specified in the bill of materials is:

Illinois Capacitor, model 108CKS100MRY

The main reason for choosing that particular part was that it was one of the shorter ones (36.5 mm) and also one of the least expensive. The height requirement was not known at the time, so shorter was a better choice. This capacitor does happen to have its ESR specified, which is 133 mΩ.

If that particular model becomes unavailable, any of the other choices would be an acceptable substitute.

3.4.9 Note on Tolerance

Nearly all aluminum electrolytic capacitors have a specified tolerance of ±20% on the capacitance value. This is a pretty wide range. It means that our 2000 µF capacitance may actually be as low as 1600 µF. Fortunately, this is more than offset by the fact that the actual sampling interval turned out to be 65 µs rather than 100 µs.

3.5 Bleed Resistor

The purpose of the bleed resistor, Rb, is to dissipate the energy from the load capacitors when they are drained between swinging IV curves. If the bleed resistor were replaced by a wire, the draining would occur very quickly and the energy would be dissipated mostly by the capacitors themselves, which could damage them or shorten their life.

3.5.1 Resistance

The resistance of the bleed resistor determines how much time it takes to drain the load capacitors. If the resistance is too high, a long waiting time will be required between curves. If the resistance is too low, it will dissipate less of the energy, and more will be dissipated by the capacitors. Ideally, the resistance should be as large as possible without requiring a longer wait time than is already required for other reasons.

Equation 3-7 below gives the voltage after draining from a voltage V_0 for time = t.

Equation 3-7: Voltage after draining for time = t

$$V(t) = V_0 \bullet e^{-t/RC}$$

One thing to realize is that the voltage approaches zero, but never actually reaches zero. In other words, the load capacitors will never completely drain even if we wait forever. But the voltage gets very close to zero very fast, and how fast (and how close) depend on the value of R (also on the value of C, but that has already been decided).

If we assume that we want to be able to swing one IV curve per second (i.e., t = 1) and we want the load capacitors to drain to 0.002% of their charged voltage (i.e., $V(t)/V_0 = 0.00002$), the only unknown is R:

$$0.00002 = e^{-1/R \bullet 0.002}$$

Take the natural log of both sides:

$$\ln(0.00002) = \frac{-1}{R \bullet 0.002}$$

Solve for R:

$$R = \frac{-1}{\ln(0.00002) * 0.002} \approx 46.2\ \Omega$$

Actually, the choice of **47Ω** for the bleed resistor was simply because I already had some when I was building the first IVS2. I used the first equation to calculate the drain level and 0.002% sounded good. It is also clearly large enough to protect the load capacitors. The only question was whether their power rating of 5W was adequate.

3.5.2 Power Rating

The energy stored by a capacitor is ½ CV^2, as given earlier in Equation 3-6. If we assume a maximum voltage of 80 V, our 2000 µF load capacitor can store 6.4 Joules as shown below in Equation 3-8.

Equation 3-8: Stored energy of 2000 µF capacitor at 80 V

$$Energy = \frac{1}{2} \bullet 0.002 \bullet 80^2 = 6.4\ J$$

Power is the rate of energy transfer:

$$Power = Energy/Time = Energy \bullet Frequency$$

If we want to swing one curve per second, frequency = 1/s. One J/s is a watt.

$$Power = 6.4\ J \bullet \frac{1}{s} = 6.4\frac{J}{s} = 6.4\ W$$

So, the 5W rating of the 47Ω resistor is not quite enough to handle continuous once-per-second IV curves with a V_{OC} of 80 V. But it is easy to argue that it is close enough. IVS2 should not be used for PV modules that have a rated V_{OC} over 80 V. But even such a module will quickly heat up, causing the V_{OC} to drop to a lower value. Furthermore, the wattage ratings of resistors are generally conservative, and are based on 24x7 usage. At worst, the resistor may get a bit hot and might eventually fail if the IVS2 is used for prolonged loop-mode tests using a max-V_{OC} PV module. This can be mitigated by rate-limiting the looping to one curve per 2 seconds.

4 Meters

There are two meters in the IV Swinger: a voltmeter and an ammeter. This section describes the requirements and design of the meters.

4.1 Meter requirements

4.1.1 Don't affect what is being measured

As is the case for all instrumentation, it is important that the presence of the meters has a negligible effect on what they are measuring (no "observer effect"). In this case that means the meters should not change the voltage or the current that they are measuring. This may sound obvious, but consider that the whole purpose of tracing an IV curve is to measure the effects of loads on the circuit. Therefore, the meters must have a negligible contribution to the load, i.e. almost no current should be diverted from the load circuit.

4.1.2 Software readability

The values read from the meters are not useful if they are only presented on a human-readable display. Software running on the computer must be able to read and record the values.

4.1.3 Accuracy and Precision

Accuracy and precision are related but different requirements.

The accuracy of the measured values is not critical when the IV Swinger 2 is used as a tool to educate students first learning about PV technology. It won't affect the learning process if the values are off by 5% or even 10%. Precision is more important. i.e. it is OK if the values are inaccurate by 5% or 10% as long as all measurements have the same offset from reality. Specifically, it is important that values measured on one run can be compared with values measured on a different run with fairly high precision. Even more important are the relative values of measurements taken on the same run. If the measured values don't have adequate significant digits, the graph will have stair steps rather than smooth sloped lines. And if the values do have adequate significant digits but the measurements don't have that actual amount of precision, the graph will be "noisy".

Accuracy is more important when IV Swinger 2 is used for research or other purposes. While this was not its original intended use, an accuracy of ±1% or less is highly desirable if it can be achieved without increasing the cost or complexity substantially.

4.1.4 Speed

The speed requirement of the IVS2 meters is much greater than it was for the IVS1 meters. If the meters are slow, a large capacitor would be required to limit the speed that the IV curve is traced out. The capacitance was chosen assuming that one (I,V) pair of measurements could be taken every 100 µs. That includes not only reading each of the meters, but also includes the Arduino processing time.

4.2 Meter Design

The voltmeter and ammeter consist of a shared Analog-to-Digital Converter (ADC) and simple circuits that drive the ADC inputs.

4.2.1 Analog-to-Digital Converter (ADC)

An ADC is an integrated circuit (IC) that translates a measured analog voltage level to a digital value that can be read by software. Even though the ammeter measures current, it does so by measuring the voltage across a resistor and applying Ohm's Law.

IVS1 uses the ADS1115 16-bit 4-channel ADC from Texas Instruments. That ADC has excellent resolution and some other nice qualities, but is far too slow for IVS2. The maximum rate is 860 samples per second, which is 1163 µs per reading. Furthermore, the I^2C bus interface is very slow.

IVS2 uses the MCP3202 12-bit 2-channel ADC. The original reason for this choice was simple: it was the ADC used in Jason Alderman's design. That was just a starting point, but it turned out to be a great choice, and there was never a need to make a change.

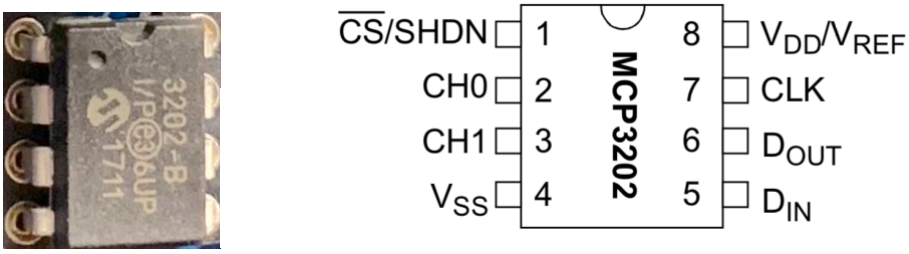


Figure 4-1: MCP3202 IC

4.2.1.1 Channels

There are two voltages that need to be measured, one for the voltmeter and one for the ammeter, so the MCP3202's two channels are sufficient. These channels are single-ended so they don't provide the noise-rejection benefit of the differential inputs that IVS1 had with the 4-channel ADS1115. That is mitigated by using low-pass filters on the inputs of the voltmeter and ammeter circuits and by performing software noise reduction.

4.2.1.2 Resolution

The 12 bits allow a resolution of 4096 (2^{12}) increments across the voltage range.

4.2.1.3 Sampling Speed

With V_{DD} = 5V, the MCP3202 has a maximum rate of 100,000 samples per second, which is 10 µs per reading.

4.2.1.4 Interface

The computer interface of the MCP3202 is the widely-used Serial Peripheral Interface (SPI) bus. The SPI interface consists of four pins: a chip-select ($\overline{CS}$), a clock (CLK), serial data-in (D_{IN}) and serial data-out (D_{OUT}).

4.2.1.5 Power / Reference Voltage

The MCP3202 works with a supply/reference voltage (V_{DD}/V_{REF}) in the range 2.7V – 5.5V. The IVS2 design uses the +5V power from the Arduino.

It is important to note that the voltage applied to Pin 8 is the reference voltage in addition to being the supply voltage, hence the name V_{DD}/V_{REF}. This means that the digital value that is returned when a measurement is taken represents a fraction of the actual voltage that is on Pin 8. The implication of this is that the accuracy of our measurements is only as good as our knowledge of the actual voltage provided by the Arduino, which is nominally +5V, but may be somewhat higher or lower[5]. For many users, this is not a significant issue, and can be ignored. For more accuracy-sensitive users, it can be mitigated by measuring the actual reference voltage when the IVS2 is connected to a given laptop; the software supports "Vref calibration". The most accuracy-sensitive users can use an AC power adaptor to supply the Arduino; this provides a regulated +5 V which is much more accurate (but less convenient) than the USB-provided +5 V. In release v2.6.0 of the software, a new feature was added that uses the Arduino 1.1 V bandgap reference to measure V_{REF} for every IV curve. This compensates for most of the variability seen when the USB-provided +5 V is used.

As mentioned in Section 3.3.4 on page 34, the electromagnetic relay draws nearly 90 mA of current when it is active. This additional load on the power supply can cause the reference voltage to "droop". This has the effect of making ADC readings higher than they should be. For the most part, this effect is factored out by calibration and software algorithms. The V_{OC} voltage is measured and calibrated when the relay is inactive, so it is not affected. However, all of the other points on the curve are measured when the relay is active, so they are affected. If V_{REF} droops, the tail of the curve does not line up with the measured V_{OC} voltage (overshoot). Software scales all of the voltages so that the tail lines up correctly. The I_{SC} current is calibrated with the relay active, so the calibration takes the reference voltage droop into account.

4.2.1.6 IC Package

The MCP3202 is available in both through-hole and surface-mount packages. The 8-pin through-hole version (8-pin DIP) is used because it may be inserted in a socket, which make it much easier to replace, or it may be soldered much more easily if a socket is not used.

4.2.1.7 Cost

The MCP3202 is very inexpensive. As of this writing, it is $2.98 from DigiKey.

[5] The ADS1115 used in IVS1 does not have this issue because it has an internal voltage regulator that provides the reference voltage.

4.2.1.8 Connections

Figure 4-2 below shows the portion of the IV Swinger 2 schematic with the connections between the MCP3202 ADC and other components.

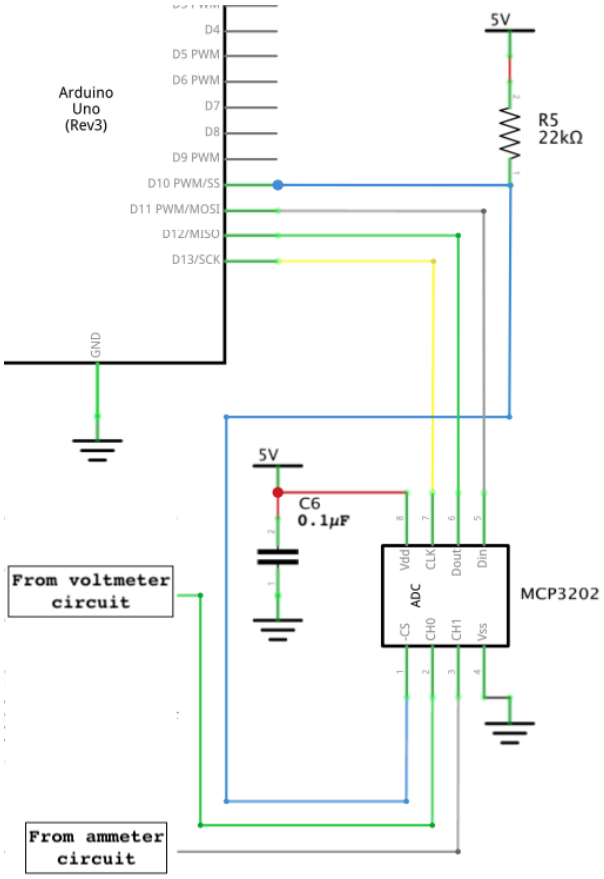


Figure 4-2: MCP3202 connections

- The V_{SS} pin is connected to ground. Ground is connected to the Arduino ground. The PV- input (black binding post) is also connected to ground. This is important because the ADC voltage measurements are relative to the V_{SS} pin and both the ammeter and voltmeter are measuring voltages that are relative to the PV- input.
- The V_{DD} pin is connected to +5V from the Arduino. It is also connected to a 0.1 µF capacitor, C6, whose other lead is connected to ground. This is the bypass (aka decoupling) capacitor specified in the MCP3202 data sheet; its purpose is to filter noise from the power supply.
- The CH0 pin is connected to the voltmeter circuit output. This is the Channel 0 input.
- The CH1 pin is connected to the ammeter circuit output. This is the Channel 1 input.
- The $\overline{CS}$ (or -CS) pin is connected to Arduino pin D10. This is the active-low chip-select, or "Slave Select" (SS) in SPI terminology. By convention (i.e. for compatibility with the SPI library code), Arduino pin D10 is always used for SS. This pin is also connected via the 22 kΩ pull-up

resistor R5 to the +5V supply. Since the chip-select is active-low, the pull-up holds it inactive when the Arduino is not driving it[6].

- The D_{IN} pin is connected to Arduino pin D11. This is the serial data-in pin, or "Master Out Slave In" (MOSI) in SPI terminology. By convention, Arduino pin D11 is always used for MOSI.
- The D_{OUT} pin is connected to Arduino pin D12. This is the serial data-out pin, or "Master In Slave Out" (MISO) in SPI terminology. By convention, Arduino pin D12 is always used for MISO.
- The CLK pin is connected to Arduino pin D13. This is the serial clock, or SCK pin in SPI terminology. By convention, Arduino pin D13 is always used for SCK.

4.2.2 Voltmeter Circuit

The maximum voltage that we need to measure is the maximum V_{OC} value of 80 V. This is much higher than the +5V ADC reference voltage, so it is necessary to scale it down. This is accomplished with a simple voltage divider circuit. As shown below in Figure 4-3, resistors R1 and R2 are in series between PV+ and PV-.

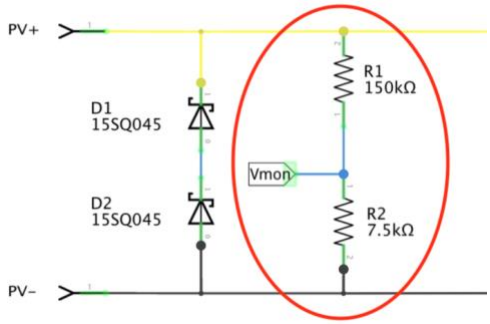


Figure 4-3: Voltmeter circuit voltage divider

The point between R1 and R2, labeled Vmon is the scaled down voltage. If the voltage is divided by too large a number, some of the resolution of the ADC would be wasted. For example, assume that it is divided by 80: the maximum V_{OC} value of 80 V would be reduced to 1 V at the Vmon point. That is only 1/5 of the maximum 5V, so instead of having 4096 voltage measurement increments, there would only be 819 (and much fewer for PV modules with lower V_{OC} values).

To have voltages between 0 V and 80 V scaled down to voltages between 0 V and 5 V, we would need a ratio of 80:5 or /16. To provide a safety margin, 100 V is a better assumption (and is consistent with the voltage limit of the load capacitors), so 100:5 or /20 was the actual target.

Equation 4-1 below is the generic equation for a voltage divider.

Equation 4-1: Voltage divider equation

$$V_{out} = \frac{R2}{R1 + R2} \cdot V_{in}$$

[6] The 22kΩ value is another vestige of the Jason Alderman design. 10kΩ is a more typical value used for a pull-up. Furthermore, it isn't entirely clear if this pull-up is necessary, but it can't hurt.

In this case, V_{in} is the PV module voltage (PV+ minus PV-) and V_{out} is the voltage at Vmon.

$$\frac{R2}{R1+R2} = \frac{7.5k}{150k+7.5k} = \frac{1}{21}$$

This is as close to 1/20 as possible with common resistor values.

The same ratio could have been achieved with R1 = 150Ω and R2 = 7.5Ω. However, at 80V the current would be V/R = 80V/(150Ω+7.5Ω) = 0.5 A. That is way too much. Not only is it a significant portion of the current generated by the PV module, the resistors would have to be huge to dissipate all of that power. Using resistances 1000 times larger avoids both of those problems. The current at 80 V is only 0.5 mA. This is negligible compared to the current being generated by the PV panel, so it satisfies the requirement that the meters not affect what is being measured.

The power rating of the resistors has to be checked too. Power is I^2R. The maximum current, as noted, is 0.5 mA, and R1 is the larger of the resistors. $I^2R = (0.0005A)^2 \cdot 150000\Omega = 0.0375W$. So ¼ W resistors are more than adequate.

The resistors used have a tolerance of ±1%. In the interest of accuracy, it is easy enough to measure their actual resistance with a multimeter before building them into the circuit and use the measured values in the software. The software supports entering the measured values under the Calibration menu.

The voltage divider output (Vmon) is not fed directly into the ADC Channel 0 input. Figure 4-4 below shows the circuit that is between these two points.

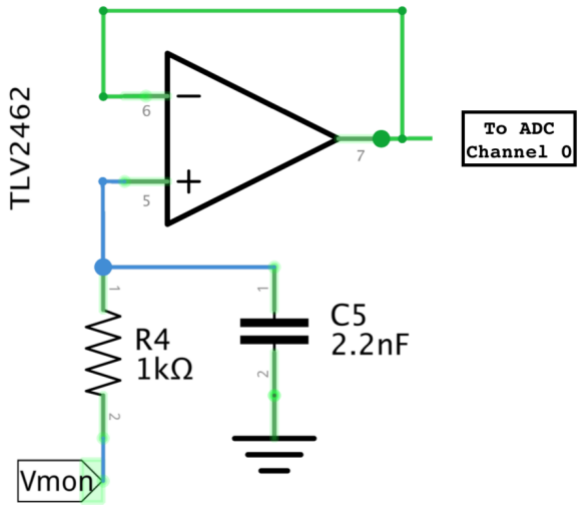


Figure 4-4: Voltmeter filter and buffer

The TLV2462 is a dual op-amp integrated circuit (IC). The other op-amp is used for the ammeter. Section 4.2.4 on page 53 describes the TLV2462 IC and why it was chosen.

With the output of the op-amp connected to its inverting (-) input, it becomes a voltage follower, i.e. the voltage at the op-amp output is equal to the voltage at its non-inverting (+) input. Its purpose is to isolate the voltage divider from the ADC input. In other words, virtually none of the current passing through the voltage divider "escapes" at the Vmon point because the op-amp input has a very high impedance. Yet

the ADC input receives all the current it needs to operate correctly from the op-amp output (which has a low impedance). A voltage follower is also known as a voltage buffer.

Resistor R4 and capacitor C5 form a low-pass filter. This reduces noise from the signal from the voltage divider. The cutoff frequency of a simple first-order low-pass RC filter is given in Equation 4-2 below.

Equation 4-2: Cutoff frequency of a first-order low-pass RC filter

$$f_c = \frac{1}{2\pi RC}$$

For R = 1kΩ and C = 2.2nF, f_c = 72 kHz. In other words, frequencies lower than 72 kHz will be passed through and frequencies higher than 72 kHz will be attenuated, with the attenuation increasing with frequency. Once again, these values were chosen because they were used in the Jason Alderman design. It is not known how effective this filtering is for the sources of noise in the actual IV Swinger 2 design, nor if it is necessary at all. It is likely that the design would work fine without this filter. The software includes a noise reduction algorithm that potentially makes this filter unnecessary even if it does provide some benefit.

4.2.3 Ammeter Circuit

There are two common ways to measure current:

- Hall-effect sensor
- Shunt resistor

A Hall-effect sensor measures the magnetic field created by the current and outputs a voltage proportional to the current. An ACS712 Hall-effect current sensor is very cheap and small and can measure up to 30A. The catch is that it only works if there are no other magnetic fields around. An electromagnetic relay violates that requirement[7].

That leaves us with the shunt resistor method. A shunt resistor is simply a very low resistance high precision resistor. By measuring the voltage drop across the shunt, the current through it can be calculated using Ohm's Law. Because of its low resistance, it dissipates little power and therefore has a negligible effect on the values being measured. Figure 4-5 below shows the shunt resistor used in the IV Swinger 2 design, and its position in the schematic.

[7] This was especially an issue for IVS1, which had 16 EMRs.

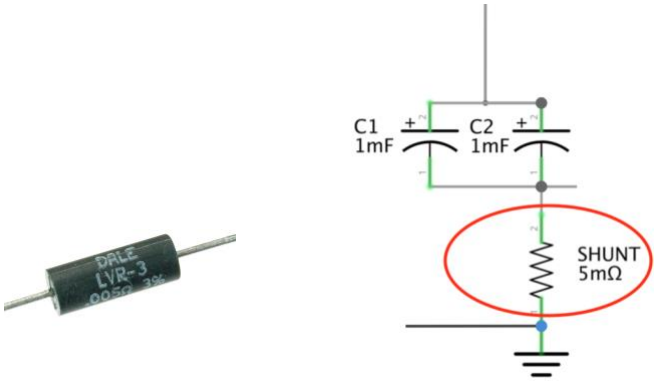


Figure 4-5: Shunt resistor

This resistor is specifically designed for current sensing. Its resistance is .005 Ω (5 mΩ). At the maximum current of 10 A, the power dissipated by the shunt resistor is I^2R = 0.5 W. It is rated at 3 W, so there is plenty of headroom. A higher resistance (up to 30 mΩ) would have worked, but minimizing the resistance of the "short circuit" load path is desirable so that the first measured point of the IV curve is at as low a voltage as possible (less error on the extrapolation of the Isc value). It is also desirable to keep the power dissipation low so that it doesn't heat up, which can affect its resistance. The only advantage of a higher resistance would have been better measurement of very low currents.

In order to measure the current through the load circuit, the shunt resistor must be part of that circuit. But where in the circuit should it go? It seems as if it shouldn't matter, since the current will be the same regardless of where it is. But since we want to measure the voltage across the shunt, it simplifies things if one end of the shunt is at the ground point in the circuit. This is known as "low-side" current sensing.

The specified shunt resistor has an accuracy of ±1%. There are alternate choices that are rated at ±3%. The accuracy rating is not really very important because there are other sources of error that require calibration for users who are sensitive to accuracy. This is also why a 4-terminal "Kelvin connection" shunt resistor is not justified.

At the maximum current of 10 A, the voltage across the shunt is V = IR = 10 * 0.005 = 0.05 V. We now have the opposite problem that we had with the voltmeter: this is 1/100 of the full ADC input range of 5 V, so the 4096 ADC increments would be reduced to only 41. The voltage across the shunt resistor needs to be multiplied before feeding it to the ADC input. This is done with the non-inverting op amp multiplier circuit shown in Figure 4-6 below.

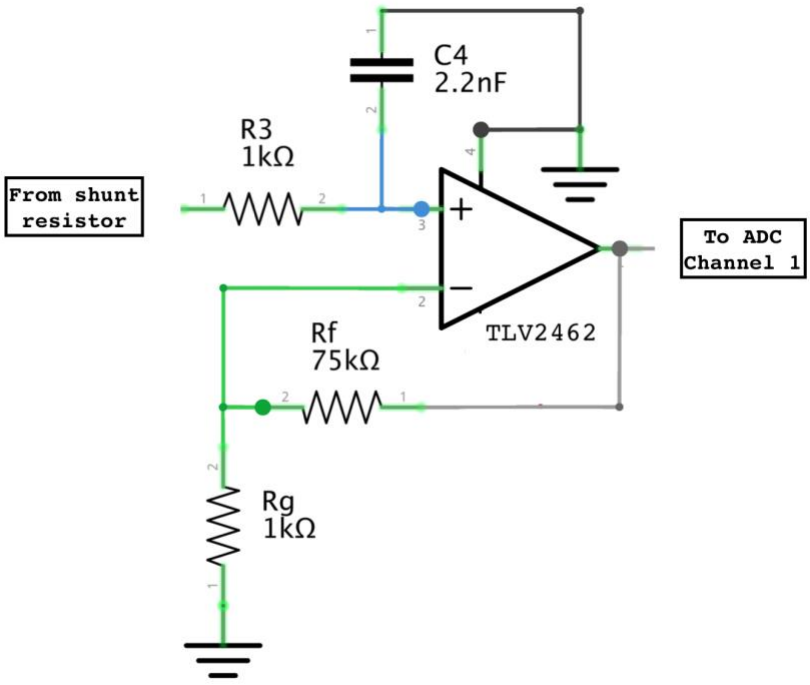


Figure 4-6: Ammeter filter and multiplier

The Wikipedia article on op amps shows how a non-inverting amplifier is built and why it works. Resistor Rg is 1 kΩ and is connected from the inverting (–) input of the op amp to ground (which is also the low end of the shunt). Resistor Rf is 75 kΩ and is connected from the op amp output to its inverting (–) input. The high end of the shunt resistor is connected to the (+) input of the op amp through resistor R3. The output of the op amp is connected to the Channel 1 (CH1) input of the ADC. The gain of the amplifier is:

$$Gain = 1 + \frac{R_f}{R_g} = 1 + \frac{75}{1} = 76$$

A current of 10A produces a voltage across the shunt of 50 mV, which is amplified to 0.05 V * 76 = 3.8 V. The ammeter maxes out (saturates) at: (5V/76)/0.005Ω = 13.16 A. A gain of 100 would have maxed out at exactly 10 A, but the chosen value gives some headroom.

Resistor R3 and capacitor C4 are the same low-pass filter circuit used for the voltmeter and discussed on page 51. Like the one in the voltmeter circuit, it is also not known how much benefit this performs. But since errors will be multiplied by 76 in this case, perhaps it is more important.

4.2.4 Op-amp IC

The TLV2462 is a "rail-to-rail" op amp IC, meaning it can generate output voltages very close to 0V at the low end and +5V at the high end. Other op amps can bottom out at values over one volt above the ground rail and can max out at values over one volt below the VDD rail (+5V in our case). This would cause inaccurate voltage and current measurements.

Two op amps are included in this IC, and it is the same 8-pin DIP package type as was chosen for the ADC (see Section 4.2.1.6 on page 47).

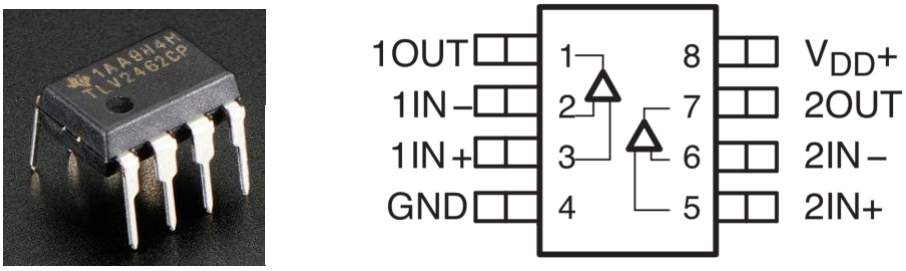


Figure 4-7: TLV2462 IC

As shown in the schematic excerpt in Figure 4-8 below:

- The GND pin is connected to ground. Ground is connected to the Arduino ground. The PV- input (black binding post) is also connected to ground.
- The V_{DD} pin is connected to +5V from the Arduino. It is also connected to a 0.1 µF capacitor, C3, whose other lead is connected to ground. This is the bypass (aka decoupling) capacitor specified in the TLV2462 data sheet; its purpose is to filter noise from the power supply.

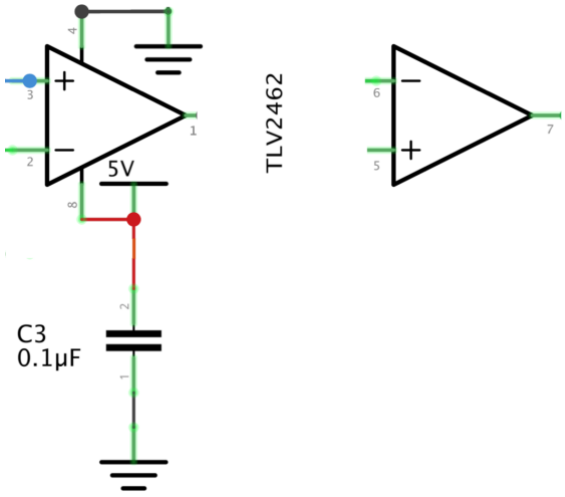


Figure 4-8: TLV2462 power and ground connections

5 Computers

Section 2.3.2 on page 23 outlines the reasons that IV Swinger 2 uses a combination of an Arduino microcontroller and a Mac or Windows laptop for the computing tasks.

5.1 Arduino

Figure 5-1: Elegoo Arduino UNO R3 clone

IV Swinger 2 uses an Arduino Uno R3 for the following:

- Receiving and processing commands from the laptop via USB
- Sending information to the laptop via USB
- Activating and deactivating the relay
- Reading the two analog-to-digital converter (ADC) channels
- Selectively storing or discarding measurements
- Permanently storing certain configuration/calibration values in EEPROM
- Reading values from optional temperature sensors
- Reading values from the optional irradiance sensor

5.1.1 Choice of Arduino

Although Arduino was an obvious choice, it was not the only option that could have satisfied the real-time requirements and been able to perform all of the necessary functions. It would have been possible to use a standalone microcontroller, as Jason Alderman did in his design. There is a reason for Arduino's enormous popularity, however, which is that the combination of the pre-installed bootloader and the Arduino IDE make it extremely easy to program. This is very important because it allows users to load and upgrade the Arduino software without any additional hardware. The fact that it is an open-source hardware platform makes the cost of the hardware so low that there is little, if any savings, from using a standalone microcontroller.

5.1.2 Choice of UNO R3

There are many varieties of Arduino. According to Arduino.cc, "The UNO is the most used and documented board of the whole Arduino family". That was the main reason for choosing it. Many of the other varieties are more expensive, larger, and have features that are not needed (faster processor, more memory, ethernet port, wi-fi, etc). The Nano might have been a reasonable choice; it is smaller but has all the same features as the UNO except for the DC power jack. The UNO turned out to be quite a nice form factor for the enclosure and for supporting the IVS2 PCB "shields" that piggyback right onto it.

5.1.3 Processor

The Arduino UNO R3 uses the ATmega328P 8-bit microcontroller, running at 16 MHz. Compared to modern microprocessors, this is very slow, but it is plenty fast for IV Swinger 2 as long as the software uses only 8-bit variables in the performance-sensitive code.

The standard UNO R3 design uses the 28-pin DIP package in a socket. There is an alternate design called UNO R3 SMD that uses a surface-mount version of the ATmega328P. The only advantage of the standard socketed version is that the processor IC can be replaced if it is damaged. But that is very unlikely and the whole board is so inexpensive that it would practically not be worth repairing. The SMD version can be less expensive and will work identically.

5.1.4 Memory

The ATmega328P has very limited, but adequate, memory.

5.1.4.1 Flash

The ATmega328P flash memory is the non-volatile memory used for the bootloader and the program (which is known as a sketch in Arduino lingo) storage. There is a total of 32 KB of flash, of which 0.5 KB is used for the bootloader and 31.5 KB is available for the sketch. The current IVS2 sketch uses only about half of this.

5.1.4.2 SRAM

The ATmega328P SRAM is the volatile memory used for the sketch's variables. There is only 2 KB of SRAM. This is quite small, but is adequate as long as the sketch is written such that memory efficiency is prioritized. The majority of the memory is used for the two arrays that store the values read from the ADC. For most IV curves, it is not possible to store every measured point, so the sketch selectively discards points that are so close to together that they don't add any value. There is enough memory to support about 275 (I,V) pairs, which is plenty.

5.1.4.3 EEPROM

The ATmega328P EEPROM is non-volatile memory that IV Swinger 2 uses primarily for saving calibration values. There is 1 KB of EEPROM, but less than 100 bytes are currently being used.

5.1.5 USB Port

The Arduino UNO R3 has a USB type B socket. A type A to type B cable is used to connect the laptop to the Arduino and provides power (see Section 5.1.8 below) and communication. The type B cable connector fits nicely through a round hole in the case, and holds the lid on.

5.1.6 Digital I/O Pins

The following Arduino digital pins are used for IV Swinger 2:

- D0: internally connected to USB port (RX)
- D1: internally connected to USB port (TX)
- D2 (output): EMR (or SSR1)
- D3 (input/output): optional DS18B20 temperature sensor
- D4 (output): 2nd EMR (or SSR5) – cell version
- D5 (output): SSR6
- D6 (output): SSR2
- D7 (output): SSR3
- D8 (output): SSR4
- D10 (output): MCP3202 -CS pin (SPI SS)
- D11 (output): MCP3202 Din pin (SPI MOSI)
- D12 (input): MCP3202 Dout pin (SPI MISO)
- D13 (output): MCP3202 CLK pin (SPI SCK)
- SCL (output): optional irradiance sensor (ADS1115 I^2C)
- SDA (input/output): optional irradiance sensor (ADS1115 I^2C)

5.1.7 Analog I/O Pins

IV Swinger 2 does not use the Arduino analog pins. When used as inputs, these pins connect to an internal 8-bit ADC. The MCP3202 would not have been necessary if 8 bits were enough resolution, but it is not.

5.1.8 Power

The Arduino UNO R3 can receive its power either from the USB port, the DC power jack, or the Vin pin. The most convenient of these is the USB port, since it has to be connected to the laptop anyway for communication purposes. The USB voltage is +5 V, but the exact voltage depends on the laptop; the Arduino voltage regulator requires at least +7 V, so it is not used when the power is coming from USB. The DC power jack and Vin pin, on the other hand, require a voltage in the range +7 V to +12 V, and that is regulated down to a very accurate +5 V. Using an external power supply is much less convenient than simply using the power from USB, however, so the majority of users will not use the DC power jack or Vin pin. Section 4.2.1.5 on page 47 describes why using external power may be desirable for very accuracy-sensitive users.

5.1.9 Cost

Arduino is an open-source hardware platform, so there are very inexpensive “clones” available. The Elegoo UNO R3 (including USB cable) is currently $11.86 on Amazon.

5.2 Laptop

Given the limited performance and memory of the Arduino, it is not used for anything that does not require real time processing. For example, it does not even convert the ADC values to current and voltage measurements; it just sends the raw 12-bit values to the laptop. The laptop runs the application software that the user interacts with.

5.2.1 Operating System

The laptop machine may run MacOS (10.10, Yosemite or newer) or Windows (7 or newer). The version restrictions apply only to the released, installable executables. In theory, older versions should be able to run the raw Python. For that matter, a Linux laptop should be able to run the Python code.

5.2.2 Hardware Requirements

There is no technical reason that the computer has to be a laptop, but given the fact that IV Swinger 2 is used outdoors, it really wouldn't make sense to use a desktop machine.

5.2.2.1 Display

The application supports scaling the display window size as small or as large as desired, so virtually any laptop display is fine.

5.2.2.2 Performance

The processor speed, memory, and disk speed will all affect the responsiveness of the application. Rendering the IV curve graph is somewhat compute-intensive, so very old and/or low-end laptops can take several seconds per IV curve. This has not been quantified, but here are two laptops for reference:

1. Dell Latitude E6400 (2010)
   - Intel Core 2 Duo p8400 2.26ghz (3MB L2)
   - 4 GB DDR2, 800 MHz
   - HDD, 7200 rpm, SATA II (50 MB/s)
   - Windows 7 Pro, SP1

2. MacBook Pro (Mid 2014)
   - Intel Core i5 "Haswell" 2.6 GHz (3 MB L3, 256 KB L2)
   - 8 GB DDR3, 1600 MHz
   - SSD, SATA III (500+ MB/s)
   - MacOS 10.14, Mojave

Laptop #1 takes 3 to 5 seconds per IV curve in loop mode. It is slow enough to be annoying, but it does work. Sometimes it takes several clicks on the "Stop" button to stop looping. Anything older or slower than laptop #1 is not recommended.

Laptop #2 can usually keep up with the minimum 1 second loop interval. Performance isn't an issue at all.

It is important to note that the actual capturing of the IV curve data is not dependent at all on the performance of the laptop since it is done by the Arduino. It is only the creation and display of the graph that taxes the performance of the laptop.

5.2.2.3 USB ports

The laptop needs at least one available USB port to connect the IV Swinger 2. This port may be USB 2.0 or 3.0. If the laptop has only USB-C ports, a USB-C to USB adapter must be used.

6 Enclosure

The main requirements of the IV Swinger 2 enclosure are:

- To protect the electronics
- To provide the interfaces to the external connections
    - Binding posts
    - USB socket
    - Temperature sensor jack (optional)
    - Irradiance sensor jack (optional)

Other objectives of the enclosure design are:

- To be aesthetically pleasing
- To visibly display the electronics
- To be easy to open and close and to replace or repair internal parts
- To be inexpensive
- To be as small as possible
- To be easy to build

The chosen enclosure is a transparent case designed for displaying collectible baseballs.

Figure 6-1: Enclosure (intended use)

6.1 Protection

The acrylic baseball case is not the most rugged choice, but it is adequate for most users. It would probably break if dropped from even a few feet high. It certainly wouldn't survive a drop from a rooftop, but neither would the laptop! A more rugged enclosure would not only have to be made of a less breakable material, but would have to have some kind of shock absorption for the electronics inside. This would be hard to justify for the ~$50 of electronics being protected.

The baseball case does effectively protect the electronics from coming in contact with anything in the outside world that could damage or disconnect wires, cause electrical shorts, etc. It also holds the components in position to protect them from each other and to keep them connected correctly.

6.2 External Connections

6.2.1 Binding Posts

Holes are drilled through the acrylic case for the binding posts. The binding posts have a backing plate that clamps the posts to the case when nuts are threaded onto the inside of the posts.

6.2.2 USB socket

A hole is drilled in the acrylic case to pass the USB type B cable connector through. This also serves as the "latch" for the enclosure; when the cable is inserted, the case is held shut and when it is not inserted, the lid opens easily.

6.2.3 Sensor Jacks

If the optional temperature and/or irradiance sensors are implemented, additional holes are drilled in the case, and mini-plug jacks are installed.

6.3 Aesthetics

The idea of a transparent enclosure was carried forward from IVS1. Aesthetics are in the eye of the beholder, but most people would agree that being able to see all of the electronics inside is way cooler than just using an opaque box of some sort.

6.4 Access

The baseball case comes in two parts, each of which is three of the six sides. The top part slides on and off the bottom part. Everything is mounted to the bottom part. As already mentioned, the USB cable holds the lid on when it is plugged in. When the lid is removed, it is easy to access all of the parts inside for repair, replacement, or probing.

6.5 Cost

Figure 6-1 above shows the Ultra Pro baseball holder listed on Amazon for $2. That is an unusually low price unless it is bought in higher quantities – it is usually around $4, but almost never over $6.

6.6 Size

The outside dimensions of the enclosure are just about perfect to hold in one hand: 3"x3"x3"

In a blatant violation of Murphy's Law, the inside dimension of the baseball case just happened to be nearly exactly the same as the length of the Arduino UNO R3, including the USB connector. The overall volume of the enclosure is enough to fit all of the electronics without being too cramped. It is actually pretty roomy for the PCB versions, especially the PV module version that is SSR-based.

6.7 Ease of Construction

Since the enclosure comes pre-built, the only construction is the drilling of holes. This takes some care, but is not difficult. The “fins” that are on the inside bottom to cradle the baseball are an annoyance, but do not have to be cut if long enough standoffs are used to mount the Arduino.

7 Hardware Design Variants

Currently there are five supported IV Swinger 2 variants. The differentiating factors are the following:

- Type of circuit board used: Perma-Proto vs. printed circuit board (PCB)
- Type of PV they work with: PV Module vs. PV Cell
- Type of relay they use: Electromagnetic (EMR) vs. Solid-State (SSR)

Table 7-1 below shows which of the combinations are supported and which are not. "Supported" means that there are step-by-step instructions and an Instructable.

	Module		Cell	
	EMR	SSR	EMR	SSR
Perma-Proto	Yes*	No	No†	No
PCB	Yes*	Yes	Yes	Yes

* Baseline design described so far in this document
† A prototype was built, however, and is depicted in the User Guide

Table 7-1: Supported Variants

Up to this point in this document, the "baseline" design that has been described applies to both the Perma-Proto and PCB versions of the EMR-based IV Swinger 2 for PV modules. This chapter describes the other variants, but only with respect to their differences from the baseline.

7.1 Printed Circuit Boards (PCBs)

The original IV Swinger 2 design used an Adafruit Perma-Proto board and hand-cut, hand-stripped, hand-soldered hookup wires for all of the connections between the resistors, capacitors, ICs, and power/ground rails. Hookup wire is also used for the connections between the Perma-Proto and the Arduino. It is still possible to build the EMR-based IVS2 for PV modules using a Perma-Proto; the documentation and Instructable still exist and the software doesn't care.

Printed circuit boards (PCBs) that provide all of these connections were developed later, once it was clear that there was enough interest in IVS2 to justify the PCB development effort. The PCBs make the construction much simpler, faster, and more mistake-proof.

7.1.1 Form Factor and Size

The IVS2 PCBs are designed as Arduino "shields". The IVS2 PCB piggybacks right onto the Arduino. This has the following desirable qualities:

- Very space-efficient
- IVS2 PCB doesn't need to be separately mounted to the enclosure
- No jumpers required between Arduino and IVS2 PCB (and no opportunity for misconnecting them)

The size and pin locations of an Arduino shield must be compatible with the Arduino. Figure 7-1 below shows the outline of the IVS2 PCBs (in red) overlaid on a drawing of the Arduino UNO R3.

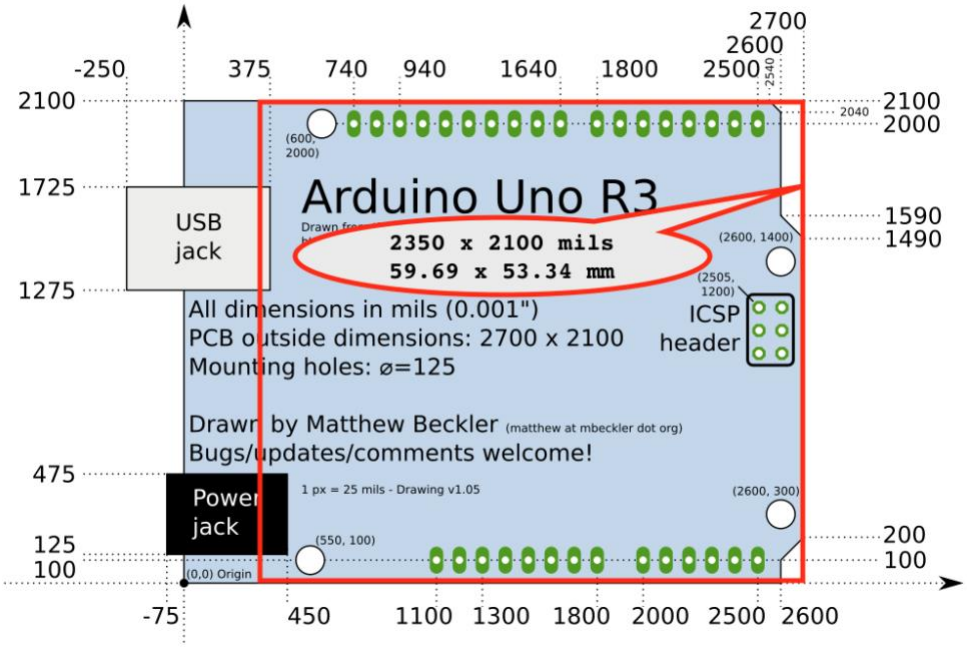


Figure 7-1: PCB form factor and size

The PCBs are all 2350 x 2100 mils, which is 59.69 x 53.34 mm. They are rectangular, without the irregular cutouts on the right end. The left end slightly overlaps the USB and power jacks, but leaves some space at that end for the wires that connect from the screw-terminal blocks on the edge of the PCB to the binding posts. Figure 7-2 below shows the unpopulated PCB for the baseline (EMR/Module) design.

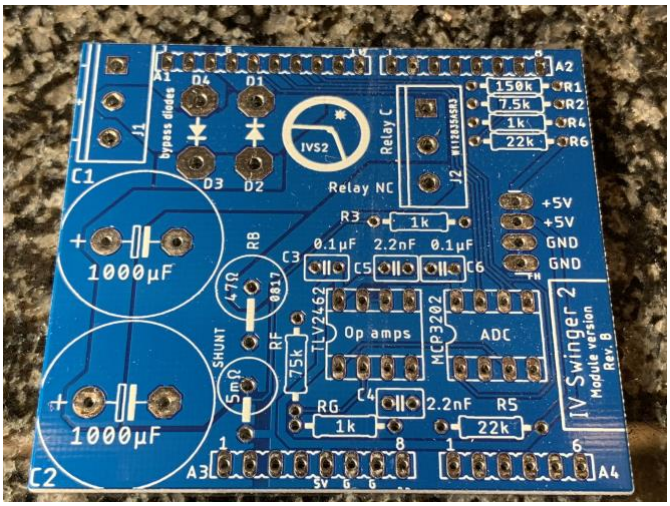


Figure 7-2: EMR/Module PCB

7.1.2 EAGLE PCB Design Software

EAGLE was used to create the four PCB designs. For small designs used for non-commercial projects, the software is free. It is well-supported and used extensively by the open-source hardware community. The primary other option was KiCad, which is open-source itself. The choice to go with EAGLE was based mostly on the fact that it appeared to me that EAGLE was more widely used. I didn't have any

prior experience with either, so that was not a factor. I really didn't spend much time choosing; either would work, so I just chose the one I'd seen more often and went with it.

7.1.3 PCB Design Files

The GitHub repository (see Section 1.1 on page 14) has a folder named "PCB" that contains the following files:

- EAGLE schematic (.sch) and board (.brd) files for each PCB variant
- PDFs of the schematic and board for each PCB variant
- Gerber (manufacturing) files for each PCB variant
- Bill of Materials (BOM) spreadsheet that can be configured for each variant
    - PDFs of BOM for each PCB variant
- GIF animations that compare the following between two variants:
    - BOMs
    - Schematics
    - Board top
    - Board bottom

7.1.4 PCB Design Considerations

The main requirement of the PCB design is to connect all of the components, on and off the board, to each other as required by the design. There are some other considerations that must be taken into account to make the board function optimally and reliably and cost as little as possible, however.

7.1.4.1 Minimizing Cost

To minimize the cost of the PCB, the following constraints are adhered to:

- Only 2 signal layers

  Using more than 2 signal layers would not only make the boards cost more, but would not qualify for the free version of EAGLE.

- Minimum trace width of 10 mils

  The wider the traces, the "sloppier" the PCB fabrication can be without resulting in bad boards. The 10-mil minimum trace width is manufacturable even by the least expensive PCB fab houses. Actually, all of the "SparkFun-2-layer-STANDARD" design rules are used, ensuring maximum manufacturability and therefore lowest cost.

- 1 oz copper

  The copper "weight", i.e. thickness, is an order-time specification. But it does affect the resistance (and therefore the current handling) of the traces. 1 oz copper is the standard, cheapest, option. The layout was created with the assumption that 1 oz copper would be used.

7.1.4.2 Minimizing Load Circuit Resistance

The resistance of the load circuit needs to be low so that it is as close to a short circuit as possible at the beginning of the IV curve. It also needs to be low in order to be able to handle the high short-circuit current without damaging the board.

Using thicker copper (e.g. 2 oz) would reduce the resistance, but would cost more. Instead, the load circuit paths are made as wide as possible by using "copper pours" instead of point-to-point traces. This can be seen in Figure 7-3 on page 67.

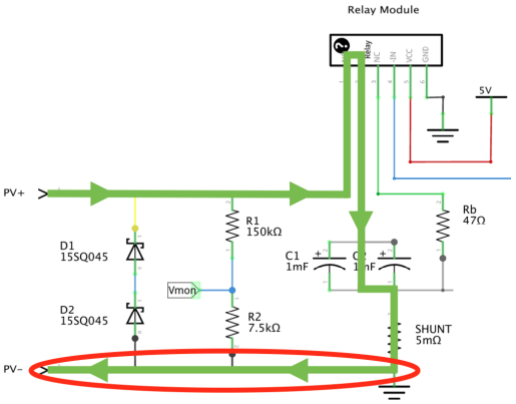
7.1.4.3 Minimizing Ground Voltage Differences

All sorts of problems can arise if "ground" is not the same voltage everywhere. Different voltages occur when current flows between any of the points that are connected to ground. The more current and/or resistance there is, the larger the voltage difference (Ohm's Law).

Using a copper pour for the ground plane reduces the resistance of these paths. The IVS2 PCBs fill all of both sides of the board (except for the load circuit pours and signal traces) with copper that connects all of the ground points.

Very little current flows in most of the ground plane. For example, the current through the voltage divider resistors R1 and R2 when the voltage is at the maximum 80 V is only 0.5 mA. The current drawn by the TLV2462 and MCP3202 ICs is similar to that. The voltage drops due to such minuscule currents is negligible.

The major exception to this is the path between the bottom of the shunt resistor and the PV- input.



That path carries the full load current of up to 10 A. The voltage drop in this path is insignificant in the context of IV curve measurements (it's just an extension of the PV- module cable). But it is much more than we want to have in a ground plane. The solution to this is to split the ground plane into two parts that are connected by a small trace at the bottom of the shunt. Figure 7-3 below shows this. All of the pink area is ground, but the whole part on the left side is isolated from the part on the right side except for the small bridge at the shunt bottom. This keeps all of the ground points in the right side from "seeing" the voltage drop in the left side.

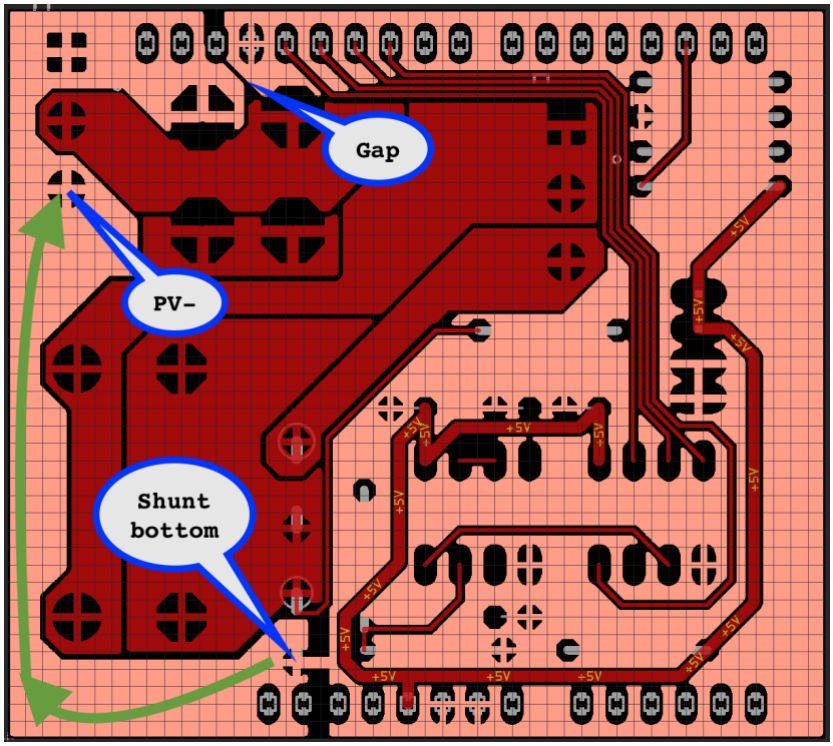


Figure 7-3: Split ground plane (top of EMR/Module PCB)

7.1.4.4 Maximizing Shunt Accuracy

The shunt resistor has a resistance of only 5 mΩ. If the voltage is measured at a point on the PCB that is connected to the top of the shunt, but is physically separated from that point with some copper trace (or pour), then the measured voltage will differ from the actual voltage across the shunt due to the resistance of the copper, which can be significant relative to 5 mΩ. This would result in a loss of ammeter accuracy.

Figure 7-4 below shows the LOAD_CAP- net. The top of the figure shows this net on the schematic. It connects the negative side of both C1 and C2 to the top of the shunt and to resistor R3, which is the ammeter input.

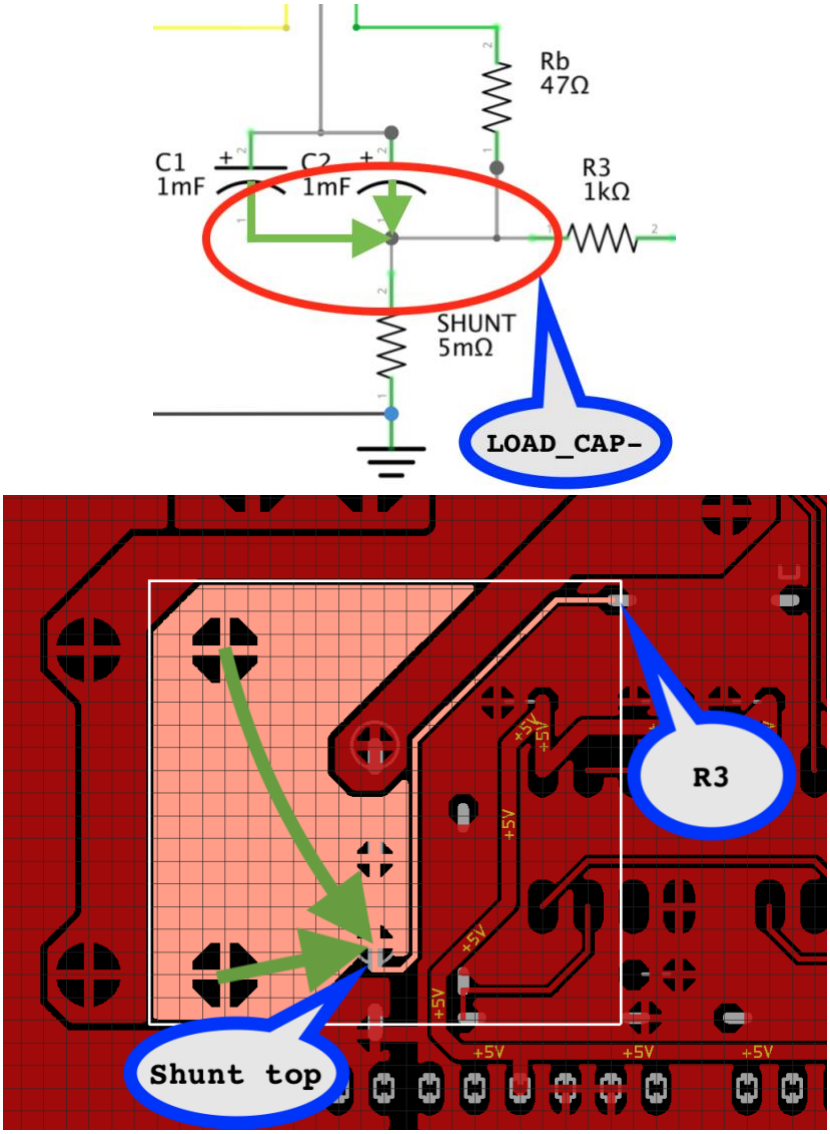


Figure 7-4: Shunt connection to R3 (LOAD_CAP- net)

The bottom of the figure shows the LOAD_CAP- net highlighted in pink. The green arrows show the current flow in the load circuit path. It is this current flow that causes a voltage gradient within the copper pour. **There is a dedicated trace that connects the shunt top to R3.** This assures that the ammeter is measuring the correct voltage and is not affected by the voltage gradient in the copper pour.

It could be argued that connecting R3 to some other point in the LOAD_CAP- copper pour would simply increase the effective shunt resistance by some constant, and that could be corrected via calibration. That may be true for the EMR-based design shown in the figure. But the SSR-based designs (Section 7.3) have two load-current paths in the same LOAD_CAP- copper pour. This means that there are two different effective shunt resistances (and only one can be calibrated). The solution is simple, but not automatic because the PCB software is “happy” as long as everything that is connected in the schematic is electrically connected on the PCB.

7.1.4.5 Minimizing Distance of Bypass Caps from IC Power Pins

The data sheets for the two ICs (TLV2462 and MCP3202) specify that the 0.1 µF bypass capacitors should be as close to their respective power pins as possible.

7.2 PV Cell Version

One might think that modifying the baseline design to work with PV cells would be easy, but it is actually very challenging.

7.2.1 PV Cell Characteristics

Silicon PV cells are available in many different sizes, but all have one thing in common which is that their open-circuit voltage (V_{OC}) is approximately the same: about 0.6 V at 25˚C and at 1000 W/m^2 of irradiance[8]. Their short-circuit current (I_{SC}), and therefore their power, is dependent on the size of the cell.

Since a typical PV module is constructed with a string of PV cells in series, we can say the following about the cells that are used to construct it:

- Cell V_{OC} = Module V_{OC} / #cells ≈ 0.6 V
- Cell I_{SC} = Module I_{SC}

Therefore, one cell from a 60-cell PV module with a rated V_{OC} of 36 V and a rated I_{SC} of 9 A will have a V_{OC} = 0.6 V and I_{SC} = 9 A.

7.2.2 PV Cell IV Curve

The IV curve of one PV cell used in a 60-cell module is the same height as the IV curve of the module, but is only 1/60 as wide. Figure 7-5 below shows a PV module IV curve and a PV cell IV curve drawn with the same voltage scale. The PV cell curve is just a sliver. It is this very small V:I aspect ratio that makes things difficult.

[8] This is for multicrystalline PV cells. It is somewhat higher for monocrystalline, but never higher than 0.764 V.

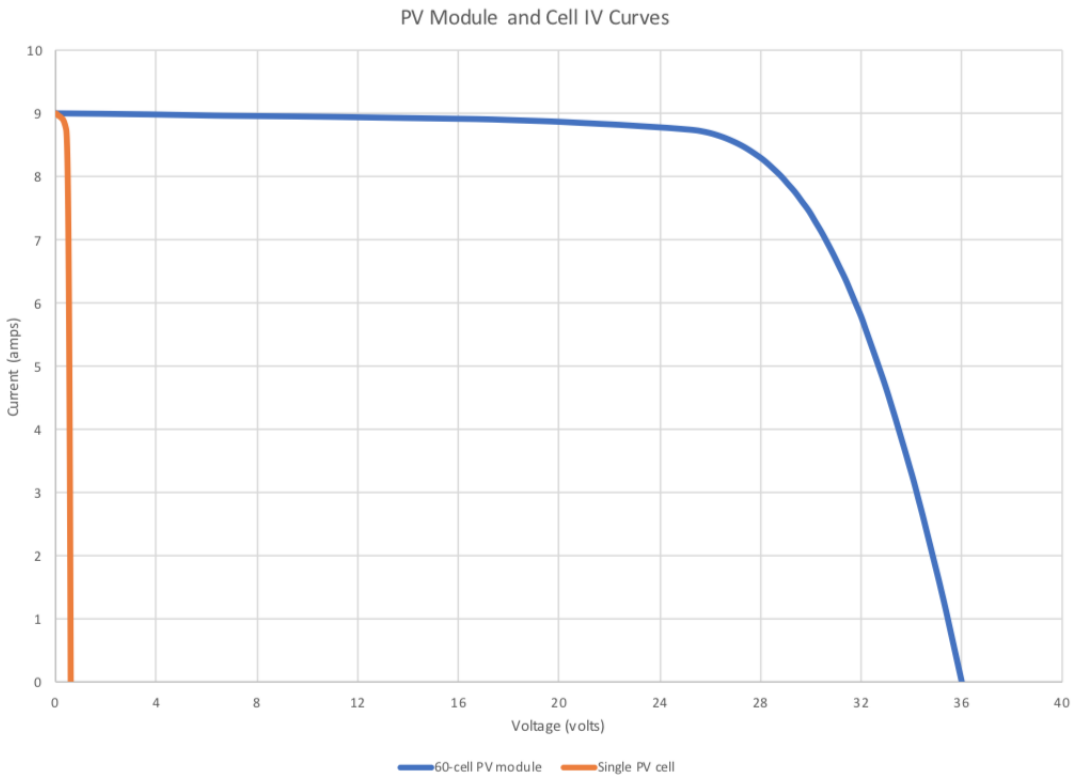


Figure 7-5: PV Module and Cell IV Curves

7.2.3 Resolution / Load Capacitor Requirements

Of course, when the IV curve for a PV cell is graphed alone, it will be plotted with a much smaller voltage range on the horizontal axis than Figure 7-5, perhaps 0 – 0.7 V instead of 0 – 40 V.

If we use the baseline IVS2 design with its two 1000 µF load capacitors, Equation 3-4 (page 36) tells us what the voltage difference would be for points along the "top" of the curve:

- C = 2000 µF = 0.002 F
- Δt = 65 µs = 0.000065 s (measured sampling rate)
- I_{avg} = 9.0 A

$$\Delta V \approx \frac{9.0\ A \bullet 0.000065\ s}{0.002\ F} = 0.29\ V$$

Given that the V_{OC} is only 0.6 V, this clearly won't work. In order to achieve comparable resolution to the PV module curve, the sampled points must be much closer together, voltage-wise, than the points on the PV module curve.

A much larger capacitance is required. 60x larger would achieve resolution parity between the 60-cell PV module and a single one of its cells. Resolution parity is not necessary, however, because the driving factor for the module IV curve resolution was the inflection points caused by the bypass diodes, and that is not relevant for PV cells.

Fortunately, the requirement for a large capacitance is offset by a much lower voltage requirement. For a single PV cell, the required capacitor voltage could be as low as 1 V. The lowest breakdown voltage for

inexpensive electrolytic capacitors is 6.3V. For reasons that will be explained shortly, this is actually a good value because it is necessary to either measure multiple PV cells in series or use a bias voltage in order to get good PV cell IV curves.

It is also desirable to find capacitors that are physically approximately the same size (diameter, height and lead spacing) as the 1000 µF capacitors that are used in the PV module design. 22000 µF capacitors are the best fit to the requirements:

Voltage: 6.3V
Capacitance: 44000 µF (2 x 22000 µF)
ESR: see next section
Package/Case: radial/can
Lead spacing: 7.5 mm (through hole)
Diameter: 18 mm
Height: 33.5 mm – 42 mm

This is 22x the capacitance of the PV module version. This gives a ΔV of less than 15 mV at 10 A, which is perfectly adequate resolution.

7.2.4 Minimum Resistance Problem

Using a larger capacitance easily solves the resolution problem. But there is another, much more challenging problem. As discussed in Section 3.4.3 on page 40, capacitors have an equivalent series resistance (ESR) which is their resistance when they are completely discharged. The wires in the load circuit path also have some resistance, as do the relay contacts and the shunt resistor. Here is a rough accounting of the minimum resistance of the load circuit path:

Wires / PCB traces	10 mΩ
Relay contacts	100 mΩ
Capacitor ESR	25 mΩ
Shunt	5 mΩ
Total	140 mΩ

Figure 7-6: Minimum Load Circuit Path Resistance

The relay contact resistance is from the Songle relay data sheet. The capacitor ESR is based on a typical value of 50 mΩ per 22000 µF capacitor, which is divided by two since the capacitors are in parallel. The total is 140 mΩ. This is pessimistic, but possible, especially as the relay ages.

Figure 7-7 below shows why this won't work. The closest point to the I_{SC} point that can be measured is well past the maximum power point (MPP) where the current is much lower than the I_{SC}. Only the tail end of the IV curve is traceable, which isn't very useful.

This could be mitigated by reducing the resistances, but it is easy to show that is a hopeless pursuit. It might be marginally acceptable for the first measurable point to have a voltage of 1/3 the V_{OC}, or around 0.2 V. That would dictate a total resistance of no more than 20 mΩ, which is simply impossible to achieve.

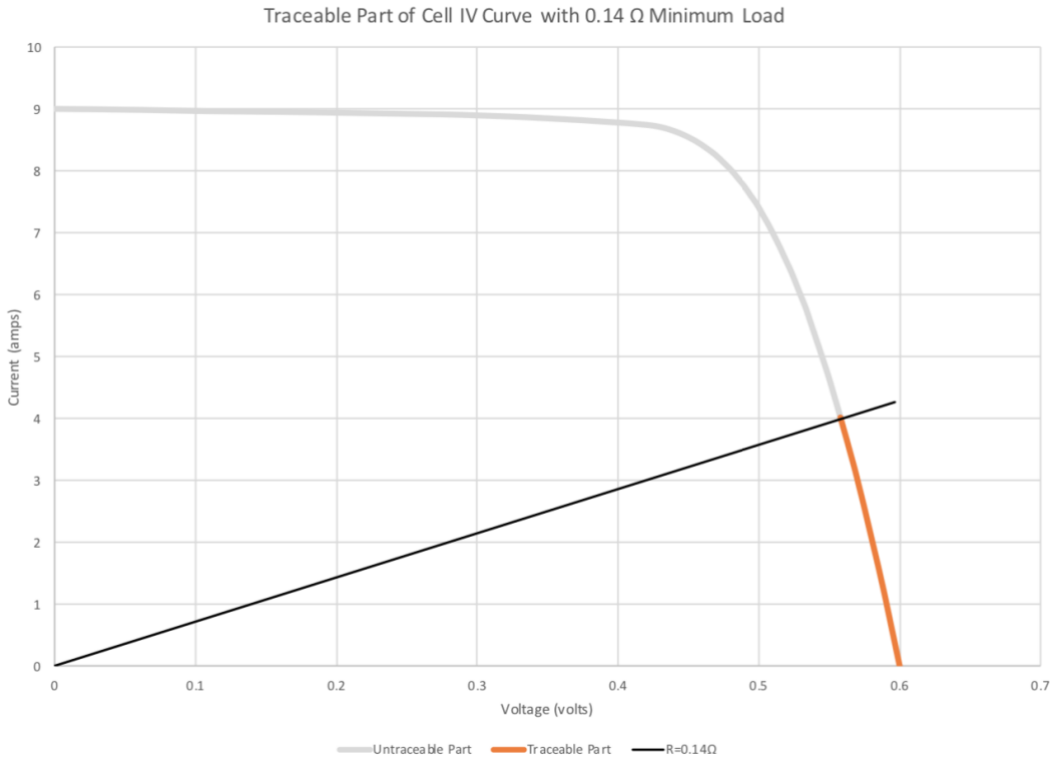


Figure 7-7: Traceable Part of Cell IV Curve with 0.14 Ω Minimum Load

There are two solutions to this problem, neither of which is ideal:

- Trace IV curves of several "identical" PV cells in series and mathematically derive the curve of a single cell
- Shift the IV curve to a higher voltage using a bias voltage source in series with the PV cell and mathematically derive the curve of the PV cell

The first solution is not ideal because the whole point of tracing IV curves of PV cells might be to compare the performance of nominally identical cells. And how do you know that they are identical if you can't compare their individual IV curves? If they aren't identical, the current will be limited by the worst one. It also costs more, takes more space, requires low resistance connections between the cells, and requires that the cells are all in identical conditions (temperature and irradiance). Essentially, this requires building a very small PV module.

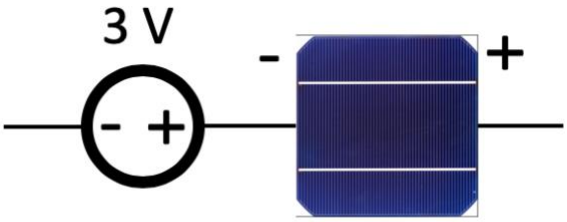
The second solution is also not ideal because it requires more hardware and because it is susceptible to small errors in the bias voltage.

Despite their shortcomings, these are the only options. The PV Cell version of IVS2 has a voltage range of 0 – 5 V, so either technique is possible. It also has hardware and software support for using a directly-connected bias battery, which makes the second technique feasible and reasonably accurate.

7.2.4.1 Bias Battery

As mentioned above, a series "bias voltage source" can be used to shift the whole IV curve to a higher voltage, which mitigates the minimum resistance problem. Figure 7-8 below shows how an ideal 3-volt bias voltage would shift the curve to the right. The current is limited by the PV cell, since it is in series

with the voltage source. There is still an untraceable part of the combination IV curve, but this doesn't matter because everything to the left of the green line will be discarded, leaving only the IV curve of the PV cell.



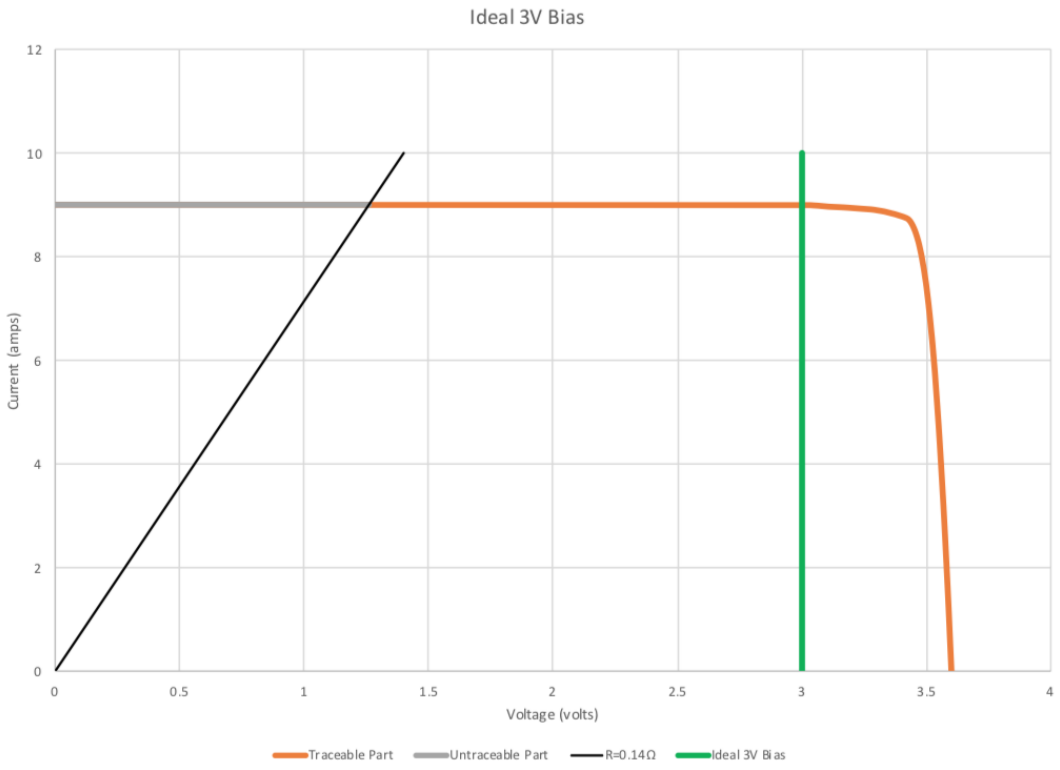


Figure 7-8: Ideal 3-volt Bias

The simplest and least expensive bias voltage sources are standard batteries. Alkaline batteries have a voltage of 1.5 V. Looking at the graph above, one might think that a single alkaline battery would be sufficient, but that is not the case. 1.5 V is the open-circuit voltage (V_{OC}). All batteries have an internal series resistance that causes the voltage to decrease pretty linearly with increasing current. The green line in the graph above represents an ideal battery with zero internal resistance. The internal resistance for real batteries is large for small batteries and smaller for larger batteries. This makes sense; a larger battery is capable of driving more current. Putting batteries in parallel effectively creates a larger battery (with the same V_{OC}).

Figure 7-9 below is the IV curve for four D-cells arranged in a 2x2 configuration. This gives a V_{OC} of 3 V with half the internal resistance of two D-cells in series (i.e. the same internal resistance as a single D-cell). The internal resistance is the negative reciprocal of the slope, or about 177 mΩ. But the important requirement is that the battery pack can deliver 10 A at a voltage of 1.4 V. That means the whole curve

will be traceable even if the minimum load circuit resistance is 140 mΩ. Figure 7-10 below shows how this works.

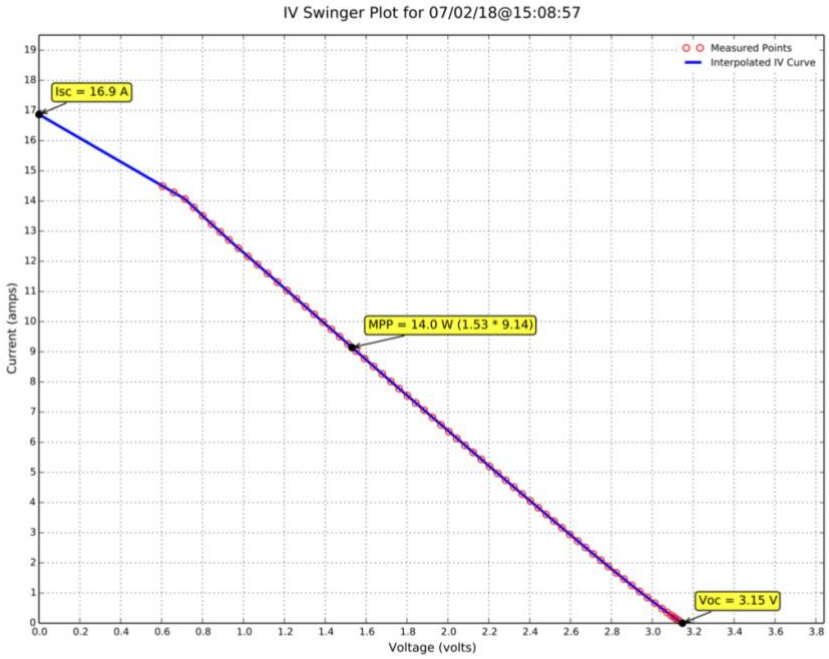


Figure 7-9: IV Curve for 2x2 D-cell Bias Battery Pack

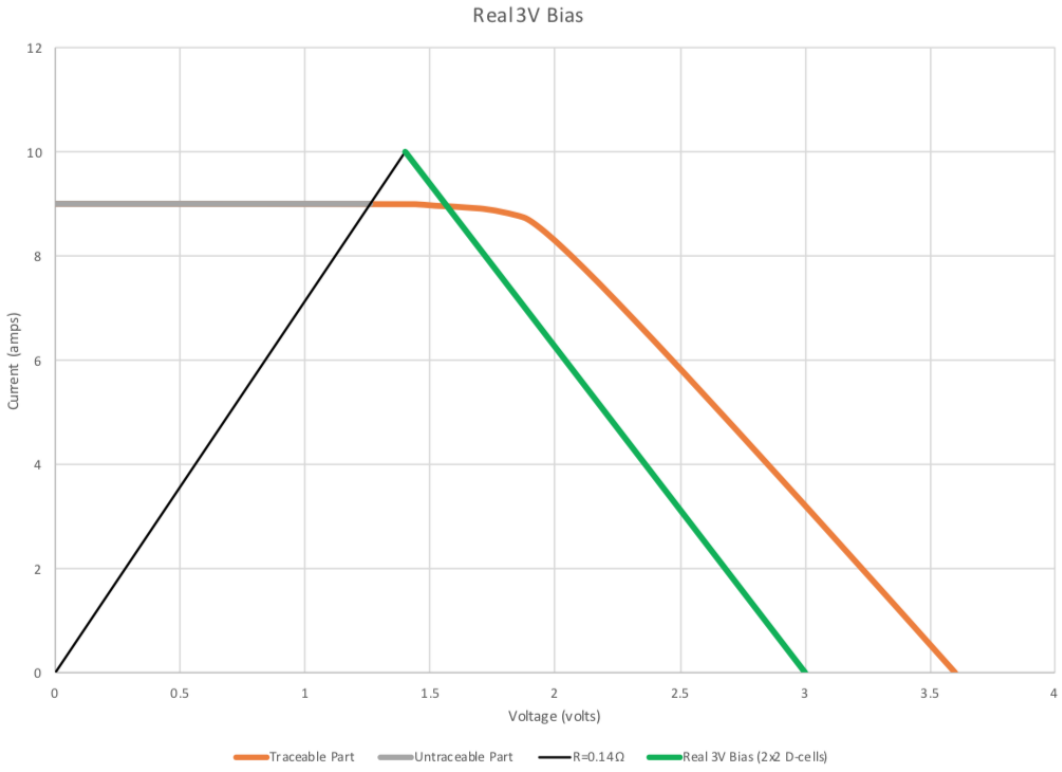


Figure 7-10: Real 3-volt Bias Using 2x2 D-cells

If the voltage bias were ideal, as in Figure 7-8 above, it would be easy to "extract" the IV curve of the PV cell; just subtract 3 V from the voltage of each measured point and discard points with negative

voltages. To extract the IV curve of the PV cell when a real battery pack is used, the voltage subtracted from each measured point depends on the current. For example, in Figure 7-10 above, the point on the orange curve at 3 V has a current of about 3 A. At 3 A, the voltage on the battery curve (green line) is about 2.5 V, not 3 A. So instead of subtracting 3 V from the measured point's voltage, 2.5 V must be subtracted. Every point on the curve has a different bias voltage.

It is necessary to know the IV curve of the battery in order extract the PV cell's IV curve from the combination IV curve of the battery and PV cell in series. Experience has shown the following:

- The battery V_{OC} and internal resistance are not 100% consistent, especially over the long term, but even in the short term
- The battery IV curve is close to being a straight line, but is actually slightly curved

For these reasons, the results are poor unless the battery curve is traced immediately before the combined curve is traced, every time. Using a linear approximation of the battery curve also produces poor results, so the actual curve must be used.

7.2.4.2 Second Relay and Second Set of Binding Posts

With a single relay and a single set of binding posts as in the baseline design, the user would have to manually connect the battery to the binding posts and swing a curve before manually connecting the PV cell in series with the battery. In addition to this being laborious, the time between the two curves could result in some inaccuracies. Instead, a second relay and a second set of binding posts are used so the bias battery and PV cell can stay connected, and the configuration change (battery only vs. battery + PV) is made automatically. The user clicks on the "Swing!" button once. The software swings the battery curve first, then it activates the second relay to put the two in series and swings a second curve. The displayed result is the PV cell curve, obtained by "subtracting" the battery curve. This is the "dynamic bias battery calibration" feature.

Figure 7-11 below shows how the second relay and binding posts are connected.

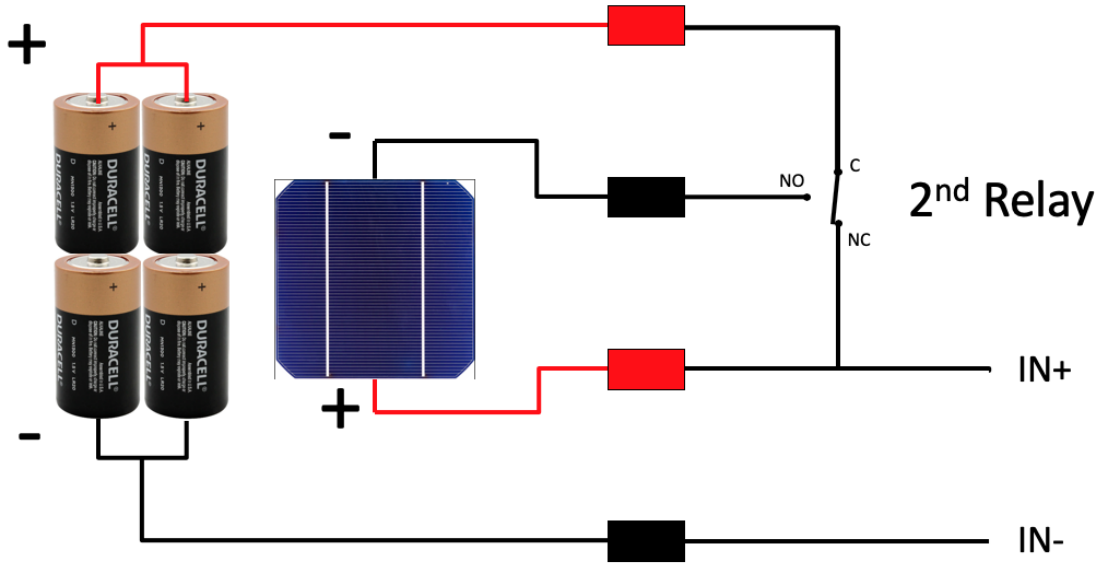


Figure 7-11: Second Relay and Binding Posts

The lines labeled “IN+” and “IN-” are the inputs to the rest of the IV Swinger 2, just as before. The second relay and binding posts are really just controlling what is going into the IVS2.

When the second relay is OFF (as shown in the figure), the bias battery pack is connected between IN+ and IN-, so the traced curve will be of the battery alone. When the second relay is ON, the bias battery and PV cell are connected in series and the combination is connected between IN+ and IN-.

7.2.5 Cell Version Schematic

Figure 7-12 below is the schematic for the EMR-based cell version of IV Swinger 2.

To more easily compare this schematic with the schematic for the baseline PV module version (Figure 2-4 on page 24), the GitHub repository contains a GIF animation that flips back and forth between the two.

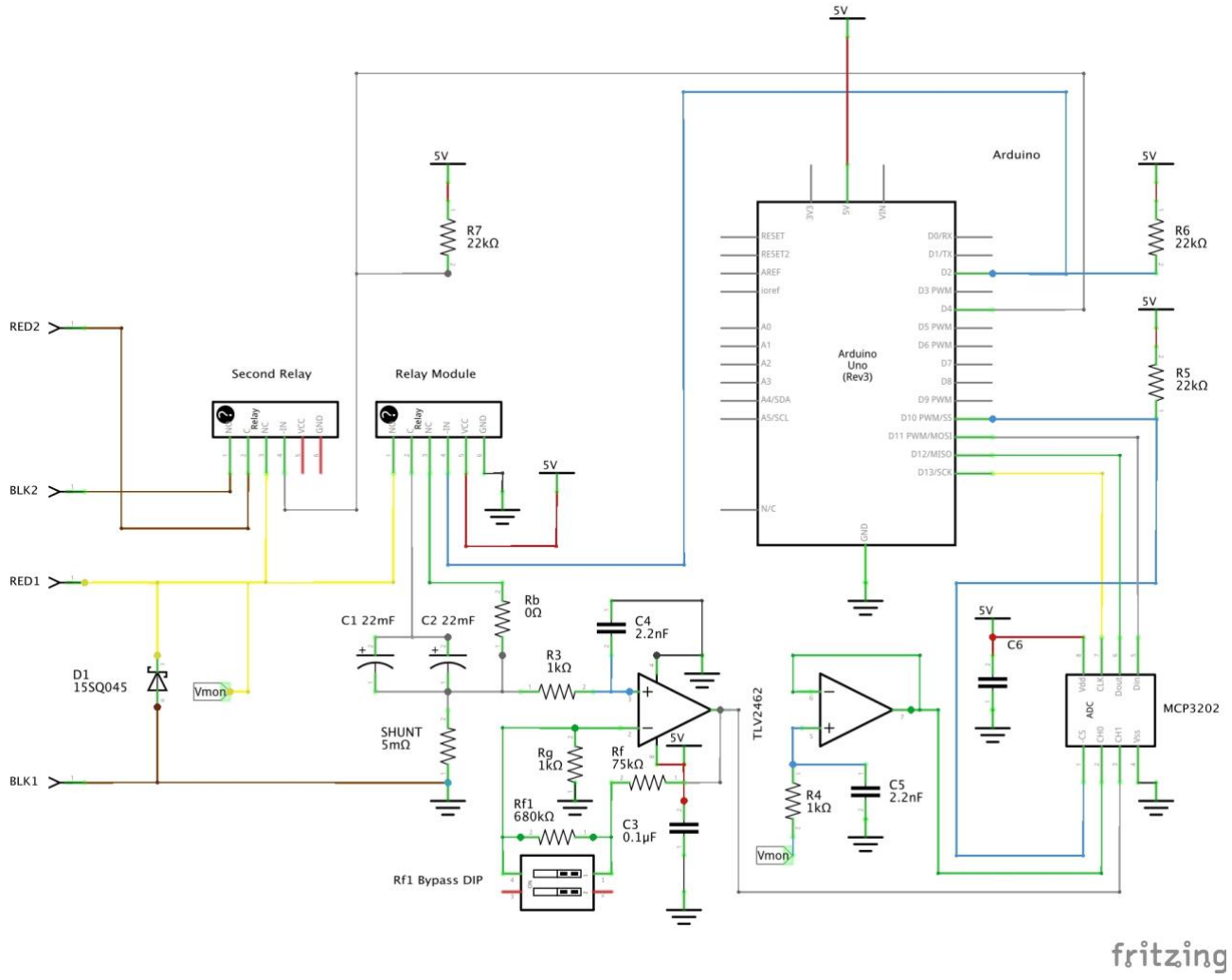


Figure 7-12: Cell Version Schematic (EMR)

The cell version has the following differences versus the module version:

- Load capacitors are 22000 µF each (vs 1000 µF)
- Second set of binding posts

- Second relay module
- No voltage divider
- No bleed resistor (Rb = 0Ω = wire)
- Ammeter multiplier has DIP switch and 680 kΩ resistor Rf1 to increase gain by ~10x
- Only one 45V bypass diode

The first three have been discussed already. The schematic shows that the second relay is controlled by Arduino pin D4. Also note that the binding posts are just named BLK1, RED1, BLK2, and RED2 since they can be connected differently depending on whether the bias battery is being used or not.

The elimination of the voltage divider gives a voltage range of 0 – 5 V, which is appropriate for the 3 V bias battery plus a 0.6 V PV cell.

The elimination of the bleed resistor is a bit questionable, but the energy stored by the load capacitors is far less than it is for the module version: 0.5 * 0.044 F * (5 V)^2 = 0.55 J. Some energy will be dissipated by the relay contacts, but the rest will be dissipated by the capacitors. Using the math from Sections 3.5.1 and 3.5.2 on page 43, there would be a 2 Ω, ½ W bleed resistor. The assumption is that the capacitors can dissipate this small amount of energy without being damaged (at least through the number of likely IV curves during the lifetime of the IVS2).

The DIP switch on the ammeter multiplier selects whether the Rf resistance will be the standard 75 kΩ (switch ON) or 75 kΩ + 680 kΩ = 755 kΩ (switch OFF). If it is the larger value, the ammeter gain will be 766 instead of 76. This is to support small PV cells that have an I_{SC} of less than about 1.3 A[9].

The single 45V bypass diode is because the maximum voltage is only 5 V. Of course, a smaller voltage diode may be used.

7.3 Solid-State Relay (SSR) Versions

The electromagnetic relay (EMR) module used in the baseline design is very inexpensive and has been shown to work quite well. But it does have some downsides:

- It is a mechanical device, susceptible to wear, degradation and eventual failure
- It requires off-board wires and mounting
- It is only rated at 30 VDC, so its lifetime may be shortened dramatically when used with higher voltage PV modules

A Solid-State Relay (SSR) is a device that can be soldered directly to a PCB. Like an EMR, it serves as a switch that can be turned on or off by a control signal. However, whereas an EMR is a single-pole double-throw (SPDT) switch, an SSR is a single-pole single-throw (SPST) switch. An SSR has no moving parts; it is a purely electrical device. SSRs are available with current and voltage ratings that exceed the advertised limits for IV Swinger 2.

[9] In practice, this doesn't work very well. A larger shunt resistor is needed for accurate measurements of low currents.

The main disadvantage of SSRs is their cost, which is exacerbated by the fact that three are needed for the module IVS2 and four are needed for the cell IVS2. Three SSRs cost nearly $20, compared to less than $2 for one EMR module.

7.3.1 How Does an SSR Work?

An SSR looks like this[10]:

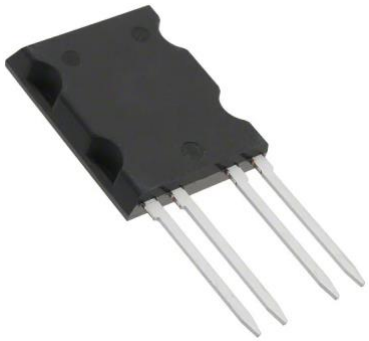


Figure 7-13 below shows the internal block diagram and what is connected to each pin.

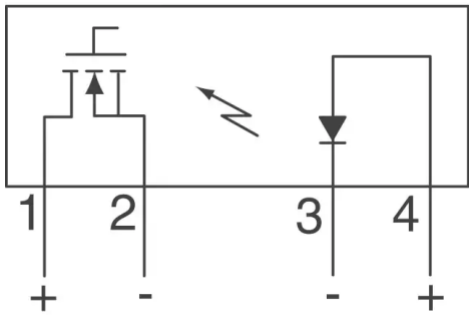


Figure 7-13: SSR block diagram

The load circuit is connected to pins 1 and 2, with pin 1 connected to the higher voltage. Internally, pin 1 is connected to the drain (D) of an N-channel MOSFET and pin 2 is connected to its source (S). The MOSFET's gate (G) is driven internally by a photodiode (not shown). Pins 3 and 4 are connected to an internal LED. If pin 4's voltage is sufficiently higher than pin 3's voltage, the LED turns on. When the LED is on, the photodiode on the MOSFET gate turns the MOSFET on, and current is allowed to flow from pin 1 to pin 2. When the LED is off, the MOSFET blocks current from flowing from pin 1 to pin 2. This arrangement provides galvanic isolation (specifically, optical isolation) between the load circuit and the control circuit.

7.3.2 Active-low vs Active-high

The external connections to pins 3 and 4 determine whether the SSR is active-low or active-high. For active-low, pin 4 is connected to the voltage source (+5 V in our case), and pin 3 is the active-low control pin (driven by the Arduino in our case). The LED will turn off when pin 3 is driven high and will turn on when pin 3 is driven low. For active-high, pin 3 is connected to ground, and pin 4 is the active-

[10] There are other form factors, but this is what the SSRs used for IVS2 look like.

high control pin. The LED will turn off when pin 4 is driven low and will turn on when pin 4 is driven high. A resistor is required to prevent sourcing or sinking more current than the Arduino pin can handle. Figure 7-14 below shows the active-low and active-high configurations, including the current-limiting resistors.

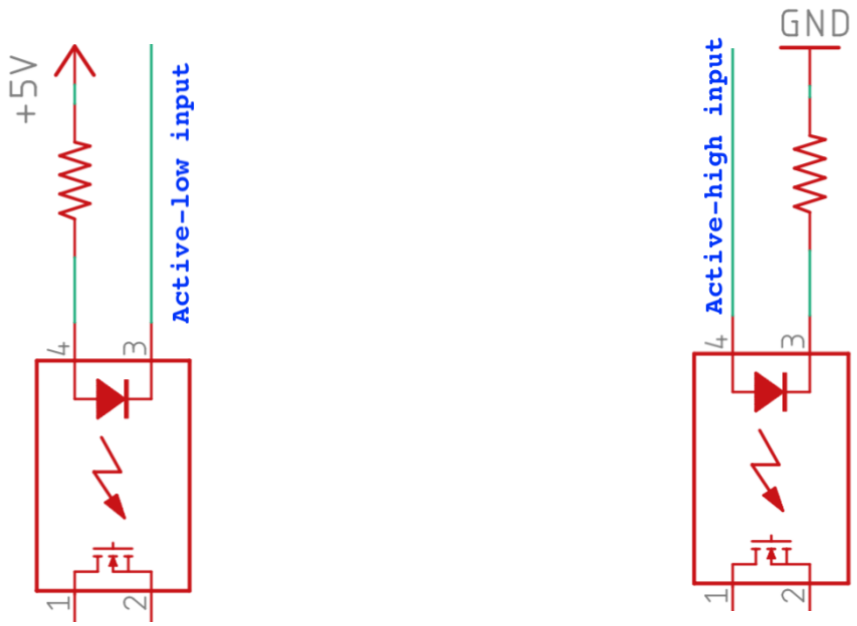
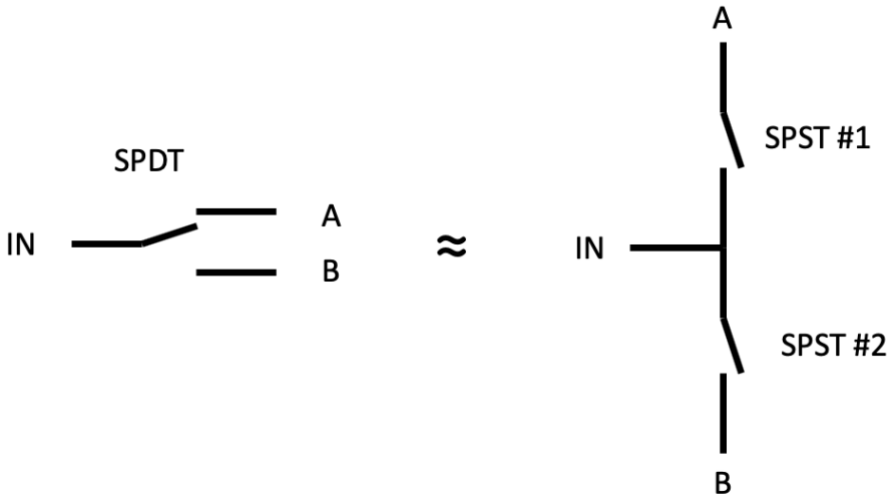


Figure 7-14: Active-low and active-high configurations

7.3.3 Using Two SSRs to Create an SPDT Switch

As previously noted, an EMR is a single-pole double-throw (SPDT) switch whereas an SSR is a single-pole single-throw (SPST) switch. So, it is not possible to simply swap the EMR out for an SSR. But it is easy to create one SPDT switch from two SPST switches, with some caveats.



	SPDT	SPST #1	SPST#2
IN connected to A	Off	On	Off
IN connected to B	On	Off	On

Figure 7-15: SPDT from two SPSTs

Figure 7-15 above shows how this is done. The caveats are:

- There are two states of the SPSTs that do not map to either of the SPDT states (Off/Off and On/On)
- There is one control signal for the SPDT and there are two control signals for the pair of SPSTs

It is important to analyze what happens in the two invalid SPST states, because even if the two SPSTs are switched "simultaneously", there most likely will be a small amount of time that they will be in at least one of those states. The IVS2 load circuit now looks as shown in Figure 7-16 below. There is no issue with both SSRs both being in the Off state (in fact even the EMR is in this state very briefly as the contact is "in the air" while switching). We really don't want both SSRs to be in the On state at the same time, because when the load capacitor is charged, all of the current will go through the bleed resistor Rb which can only handle 5 W.

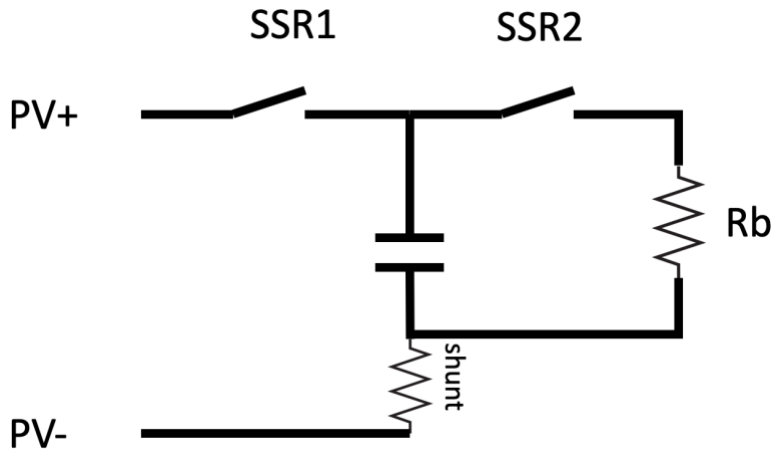


Figure 7-16: Basic Load Circuit with SSRs

Since we always want SSR1 to be On when SSR2 is Off and vice versa, it would seem easy to control both with the same signal by configuring SSR1 as active-low and SSR2 as active-high as shown below in Figure 7-17.

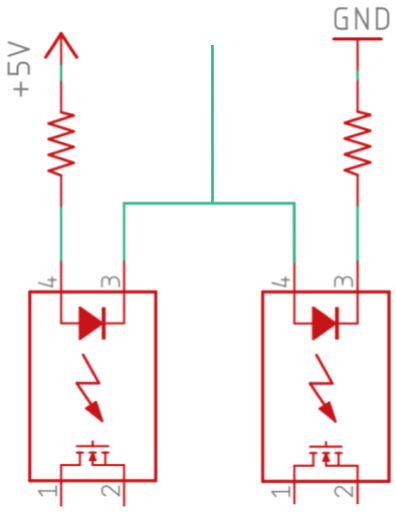


Figure 7-17: Single Control Signal for both SSRs?

There is a problem with doing that, however. Note that there is a path from +5 V to GND, as shown in Figure 7-18. If current flows in this path, both SSRs turn on, which is the case we want to avoid since it directs all of the PV current through the bleed resistor Rb. This can happen if the control signal is "floating", i.e. if the Arduino pin is not driving it high or low, which is the case before the software configures the pin as an output. Even putting a pull-up resistor won't necessarily solve the problem. Once it gets into this state, it tends to stick there which not only can burn out Rb, but is more current than the SSRs can handle on a continuous basis[11]. There are potential ways to make this work, but it is much safer to simply use a separate Arduino pin for each SSR. This breaks the path shown in Figure 7-18 and it also allows the software to control the order so the "both on" case can be avoided.

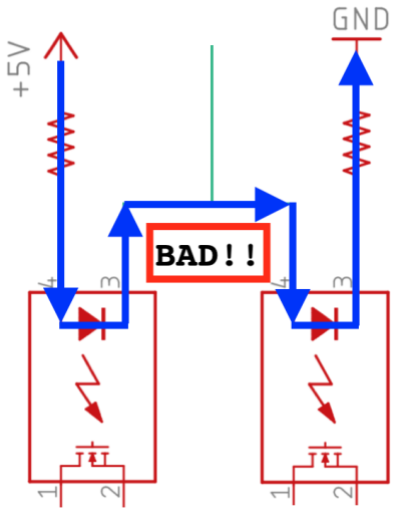


Figure 7-18: Path from +5V to GND

7.3.4 SSR Requirements

The voltage and current handling capabilities of the SSRs must meet the needs of IVS2. The resistance of the load circuit path and switching speed are also relevant.

7.3.4.1 Blocking Voltage

SSRs are rated for the voltage that they can "block", i.e. the maximum voltage that can be applied between the two load circuit pins when the SSR is in the Off state. In order to be able to handle the maximum advertised V_{OC} of 80 V, an SSR is required that has a blocking voltage of greater than 80 V. Choosing the same voltage as the load capacitors, which is 100 V, makes sense and gives some safety margin.

7.3.4.2 Load Current

SSRs are also rated for how much load current they can handle. However, this is not a single number. There is a peak current rating that cannot be exceeded even for a very brief time. There are also several ratings for continuous current, depending on how hot the SSR is allowed to get (i.e. whether a heat sink

[11] This is not just a thought experiment; I made this mistake the first time around.

is used, and if so, how big). There also is a rating curve that specifies the maximum time that a given amount of current may be allowed to flow before allowing the SSR to cool off again.

For space reasons, we don't want to use heat sinks, so an SSR is needed that can handle 10 A without one. The lazy choice would be to find an SSR that can handle 10 A of continuous current with no heat sink. But this would be overkill, and would be much more expensive and larger than necessary (not to mention difficult to find). The reason that it would be overkill is that the current that flows through the SSRs is not anywhere near continuous. A typical IV curve takes less than 15 ms to swing, and some of that is on the tail of the curve where the current is lower. The minimum interval between IV curves (enforced by the software) is 1 second. For this reason, the peak current rating is more relevant than any of the continuous current ratings. A peak current rating of 20 A is almost certainly adequate. A more accurate analysis can be performed based on a given SSR's specs, however, and we'll do that in Section 7.3.5.2.

7.3.4.3 On-Resistance

Another rating included in the data sheet for SSRs is the resistance to current flowing between the load circuit pins when the SSR is in the On state. This is relevant for the same reason the contact resistance is relevant for the EMR design; it is part of the "short circuit" resistance of the load circuit (along with capacitor ESR, shunt resistance and wiring resistance). Although it is hard to determine a specific value that would be too high, we would certainly like the worst case to be less than or equal to the worst-case EMR contact resistance, which is 100 mΩ.

7.3.4.4 Switching Speeds

SSRs do not turn on or off instantaneously when the voltage across the LED is changed. The specification lists typical and maximum times for turn-on and turn-off (4 values). It is important to realize that these times are not just delays before the SSR instantaneously turns on or off. The turn on and turn off are spread out over these times, i.e. the resistance *ramps down* from open circuit to the on-resistance when turning on and *ramps up* from the on-resistance to open circuit when turning off. An EMR does not have this ramp; its contacts are either open or closed. The SSR turn-on ramp is problematic because it means that the load capacitor starts charging up before the SSR resistance has dropped to its minimum. This results in the loss of the beginning of the IV curve. If the turn-on time were on the order of one Arduino sampling interval (65 µs), this could be ignored. But the SSR turn-on times are orders of magnitude longer than that. This is the reason that SSR3 is required (as discussed in Section 7.3.6.1.1 on page 87). SSR3 makes the turn-on time non-critical, and the turn-off time is non-critical even without SSR3.

7.3.5 Chosen SSR: CPC1718

The SSR chosen for IVS2 is:

IXYS CPC1718

7.3.5.1 Blocking Voltage

The CPC1718 has a blocking voltage of 100 V, meeting the requirement.

7.3.5.2 Load Current

The CPC1718 datasheet has several specifications of the maximum allowable load current.

Load Current, T_A=25°C:		
With 5°C/W Heat Sink	17.5	A_{DC}
No Heat Sink	6.75	

Figure 7-19: CPC1718 Load Current (from "Characteristics" table)

Figure 7-19 above is from the table at the very beginning of the datasheet. It tells us that this SSR won't be able to handle 10 A of *continuous* current without a heat sink. But given that it can handle 17.5 A with a heat sink, we already can be pretty sure it will be able to handle an *intermittent* 10 A at some duty cycle without a heat sink.

Parameter	Conditions	Symbol	Minimum	Typical	Maximum	Units
Output Characteristics						
Load Current [1]						
Peak	t≤10ms	I_L	-	-	40	A_P
Continuous	No Heat Sink				6.75	A_{DC}
Continuous	T_C=25°C				32	
Continuous	T_C=99°C	$I_{L(99)}$			8.5	

Figure 7-20: CPC1718 Load Current (from "1.2 Electrical Characteristics @25˚C")

Figure 7-20 above has more information from the table on page 2 of the datasheet. It tells us that the CPC1718 can handle a whopping 40 A if the duration is 10 ms or less. Since it only takes about that long to swing an IV curve, this is a good indication that this SSR isn't going to have a problem at all with PV modules that have an I_{SC} of 10 A. It also tells us that it can even handle a continuous 32 A if the package is actively cooled to keep it at 25˚C. Clearly, the current limit is highly dependent on how hot the device is allowed to get.

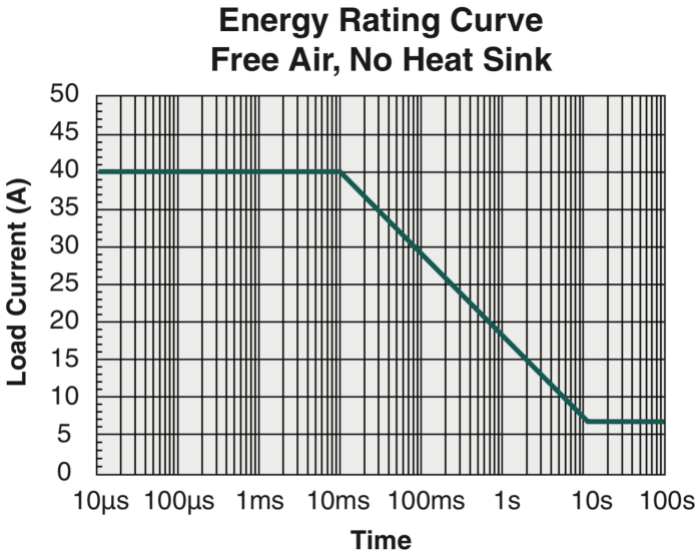


Figure 7-21: CPC1718 Load Current vs Duration

Finally, there is the chart in Figure 7-21 above, from page 5 of the datasheet. This chart shows the maximum duration allowable for different load currents when there is no heat sink. This is a very useful chart. It says that 10 A is tolerable for up to 5.5 seconds. What it doesn't tell us is how much time has to pass between each such 5.5 second period, but we can calculate that.

With no heat sink, the maximum is 6.75 A continuous. The maximum on-resistance is 0.075Ω. This tells us how much power the case alone can dissipate:

$$\text{Max continuous power} = I^2R = (6.75\ A)^2 * 0.075\ \Omega = 3.4\ W$$

If the current is intermittent, however, the device cools during the off times. The power is scaled by the duty cycle:

$$\text{Intermittent power} = I^2R * \text{duty\_cycle} = 3.4\ W$$

As long as the on-time durations do not exceed the limits in the table in Figure 7-21, the case temperature will be the same regardless of whether the power is intermittent or continuous. In other words, the case can dissipate an average 3.4 W of intermittent power just as well as it can dissipate 3.4 W of continuous power. This means the current can be higher as long as the duty cycle is lower.

Equation 7-1: CPC1718 maximum duty cycle @ 10 A

$$duty\_cycle_{max} = \frac{P}{{I_{max}}^2 * R} = \frac{3.4\ W}{(10\ A)^2 * 0.075\Omega} = 0.45$$

Equation 7-1 says that 10 A is tolerable as long as it is flowing only 45% of the time. And Figure 7-21 says that the on-times must be 5.5 seconds or less. In other words, 10 A can flow for 5.5 seconds, as long as that is followed by 6.7 seconds of zero current before the next 5.5 seconds of 10 A (and so on).

Since the IV Swinger 2 software limits the rate of swinging IV curves to one per second, the duty cycle is far less than 45% (more like 1%). We can conclude that the CPC1718 can handle the load circuit current requirements of IVS2 with a large safety margin.

Furthermore, since the CPC1718 can handle 10 A for up to 5.5 seconds, it is possible to exploit that fact to implement advanced current calibration.

7.3.5.3 On-Resistance

Figure 7-22 below is the on-resistance specification from page 2 of the CPC1718 datasheet.

Parameter	Conditions	Symbol	Minimum	Typical	Maximum	Units
On-Resistance [2]	I_F=10mA, I_L=1A	R_{ON}	-	0.03	0.075	Ω

Figure 7-22: CPC1718 On-Resistance

The worst-case is 75 mΩ. This is less than the EMR worst-case of 100 mΩ. However, as will be discussed in Section 7.3.6.1.1 on page 87, the short-circuit current must actually pass through both SSR1 and SSR3, so the total worst-case is 150 mΩ. However, that is mitigated by the fact that the typical on-resistance is only 30 mΩ, and values larger than 34 mΩ are very rare as shown in the chart in Figure

7-23 below. Unlike the EMR contact resistance, the SSR on-resistance does not degrade with use. Furthermore, since SSR3 bypasses the load capacitors, its on-resistance takes the place of the load capacitors' ESR.

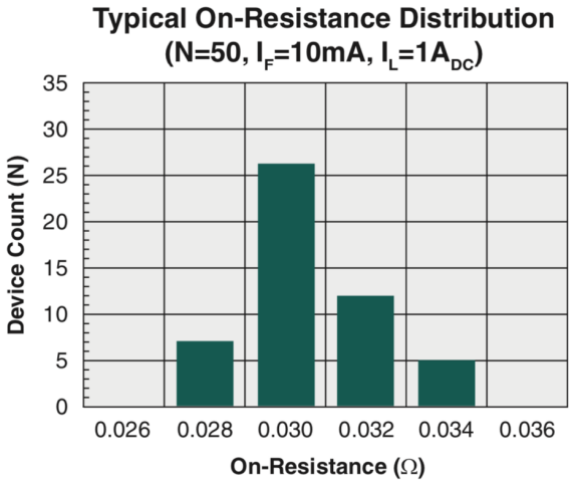
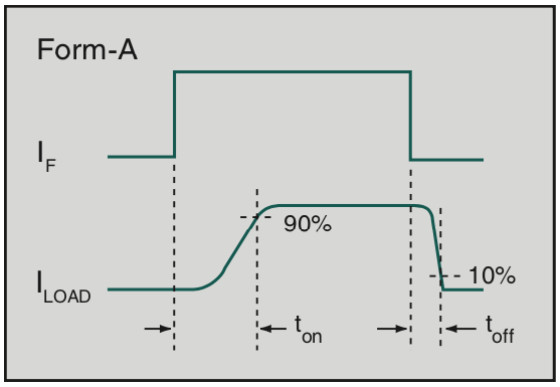


Figure 7-23: CPC1718 Typical On-Resistance Distribution

7.3.5.4 Switching Speeds

The CPC1718 switching speeds are shown in Figure 7-24 below. They are defined relative to the current (I_F) being applied through the LED (shown as a square wave). Note that "on' is only 90% on and "off" is only 90% off (10% on).



Parameter	Conditions	Symbol	Minimum	Typical	Maximum	Units
Switching Speeds						
Turn-On	I_F=20mA, V_L=10V	t_{on}	-	7.5	20	ms
Turn-Off		t_{off}	-	0.19	5	

Figure 7-24: CPC1718 Switching Speeds

The maximum turn-on time is 20 ms. That is about twice as long as it takes to swing a typical IV curve. Even the typical turn-on time is a very slow 7.5 ms. For this reason, it is not possible to simply replace the EMR with two SSRs as shown in Figure 7-16 on page 80.

7.3.5.5 LED Forward Current and Voltage Drop

On the control side of the CPC1718, the datasheet specifies how much forward current must flow through the LED to activate the SSR and how low the current must be to deactivate it. It also specifies the voltage drop across the LED when it is on. Figure 7-25 below is from the table on page 2 of the datasheet.

Parameter	Conditions	Symbol	Minimum	Typical	Maximum	Units
Input Control Current to Activate [3]	I_L=1A	I_F	-	-	10	mA
Input Control Current to Deactivate	-	I_F	0.6	-	-	mA
Input Voltage Drop	I_F=5mA	V_F	0.9	1.2	1.4	V

[3] For applications requiring high temperature operation (> 60°C) an LED drive current of 20mA is recommended.

Figure 7-25: CPC1718 LED Current and Voltage

The graphs in Figure 7-26 below are also useful.

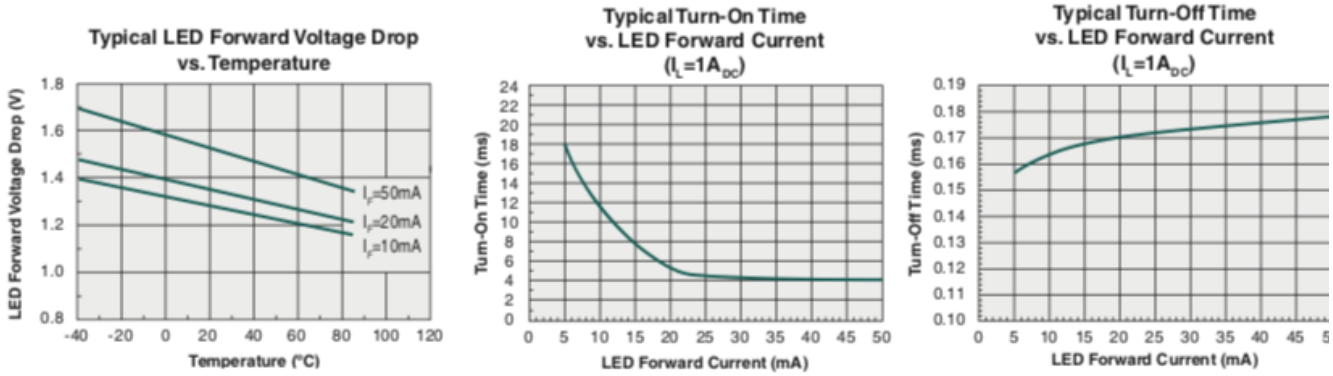


Figure 7-26: CPC1718 LED Voltage and Current Graphs

When configured as outputs, the Arduino's digital pins can source or sink up to 40 mA each, with 20 mA being the recommended design limit. 20 mA is a good target since it minimizes the turn-on time. The resistors in Figure 7-14 on page 79 are used to limit the current seen by the Arduino pins. With a 5V supply voltage, and typically 1.3 V across the diode (first graph), the resistor value is given by Equation 7-2 below.

Equation 7-2: CPC1718 Current Limit Resistor Value

$$Resistor\ value = \frac{5\ V - 1.3\ V}{0.02\ A} = 185\ \Omega$$

180 Ω is the closest standard value.

7.3.6 SSR-Based IV Swinger 2 Circuit Designs

SSR-based IV Swinger 2 PCB designs are available for both the PV module and PV cell versions. They are very similar to their EMR-based counterparts in all respects other than the switching.

7.3.6.1 SSR Version for PV Modules

The PV module version uses three CPC1718 SSRs. Two are required simply to take the place of a single SPDT EMR, as described in 7.3.3 on page 79. The third is required to deal with the slow turn-on time of the CPC1718.

7.3.6.1.1 SSR3, the Solution to Slow Turn-On

By adding a third SSR, the turn-on time becomes unimportant. Figure 7-27 below shows where SSR3 is placed in the load circuit.

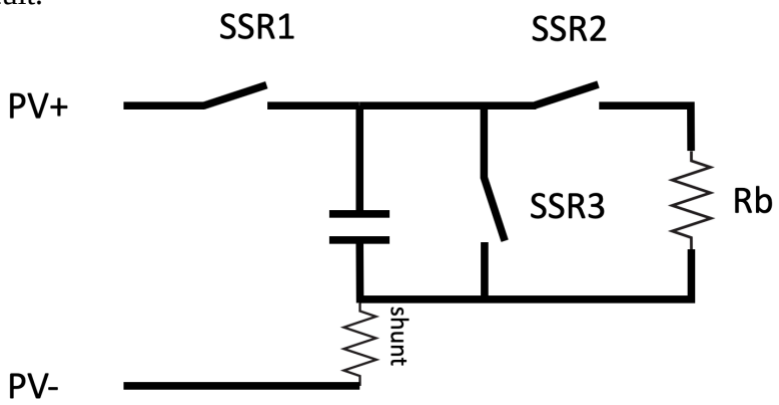


Figure 7-27: SSR3, the Solution to Slow Turn-On

There is a GIF animation in the GitHub repository that shows the sequencing. Figure 7-28 on the next few pages contains the frames from that animation, but they are pretty small, so viewing the animation is recommended.

The purpose of SSR3 is to provide a short circuit path that bypasses the load capacitors. The starting states of the SSRs are (OFF = open, ON = closed):

SSR1 = OFF
SSR2 = ON
SSR3 = OFF

The V_{OC} is measured in this state. The Arduino code then turns SSR3 ON, and waits for 20 ms. Then it turns SSR1 ON and SSR2 OFF. Now the SSRs are in the following states:

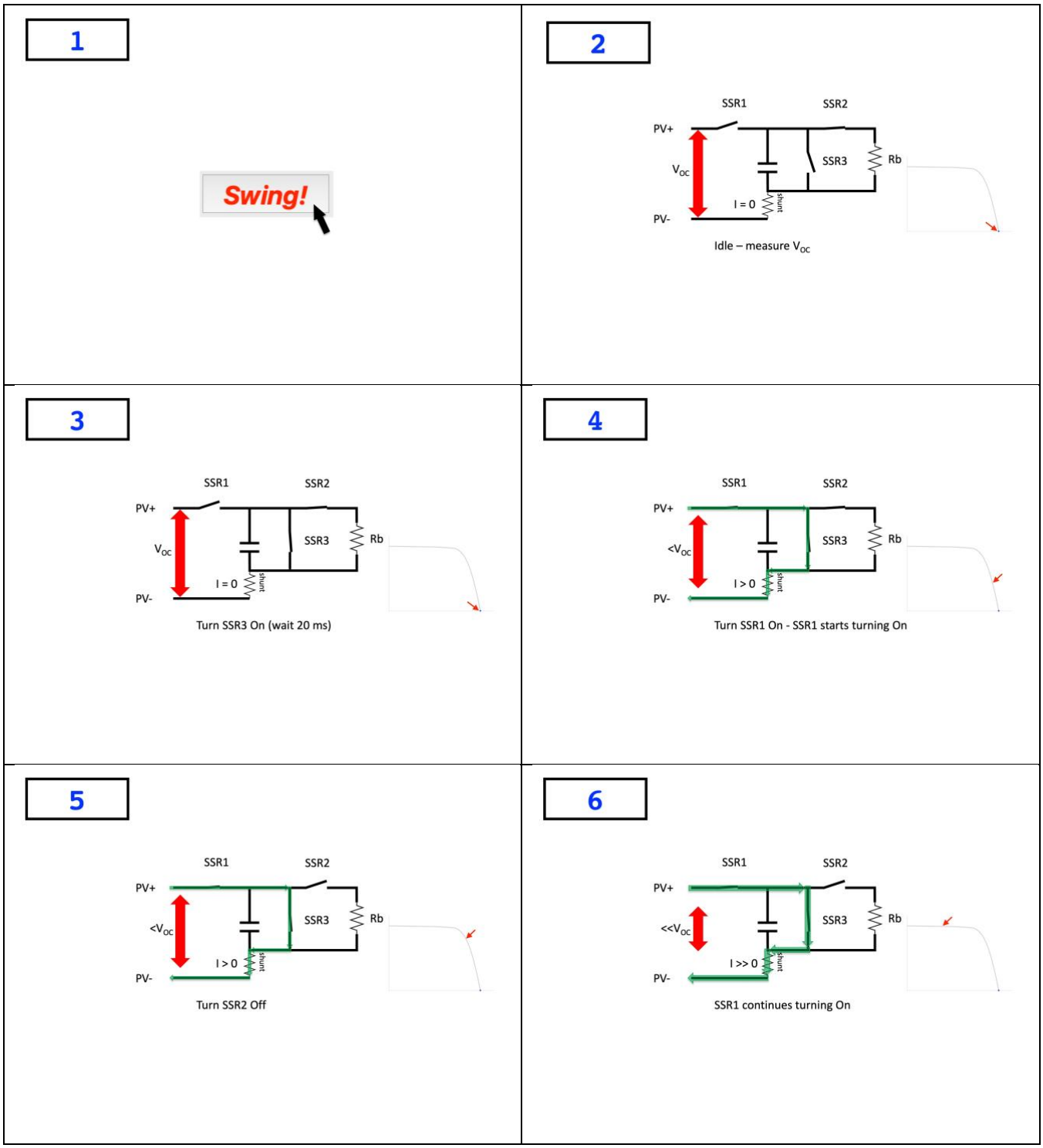
SSR1 = ON
SSR2 = OFF
SSR3 = ON

The capacitors are still discharged. There is a path through SSR1 and SSR3 around the capacitors.

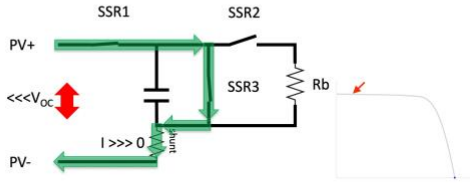
Due to the slow turn-on time of SSR1, the voltage does not drop from Voc immediately to 0, and the current does not immediately rise from 0 to I_{SC}. While SSR1 is in the process of turning on, the measurements are following the IV curve backwards, as shown in the animation. The Arduino code polls the current and voltage measurements, waiting for a stable I_{SC} value (voltage has stopped decreasing and current has stopped increasing). It discards all of the points before detecting the stable I_{SC} condition.

When the stable I_{SC} is detected, SSR3 is turned OFF. Even though SSR3 takes time to turn off, it doesn't affect the IV curve at all. The points that are measured while SSR3 is in the process of turning off are completely valid. It is irrelevant that a small amount of current is going through SSR3 instead of through the load capacitors.

Note that it would be possible to achieve the same thing without SSR3 if Rb were a 0Ω resistor (wire). In fact, the SSR-based cell version does exactly that (otherwise it would require 5 SSRs instead of 4).

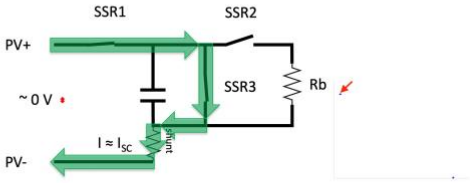


7



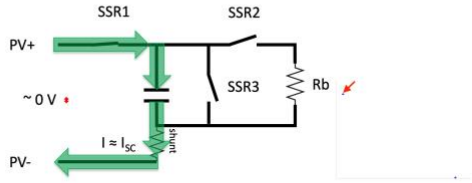
SSR1 continues turning On

8



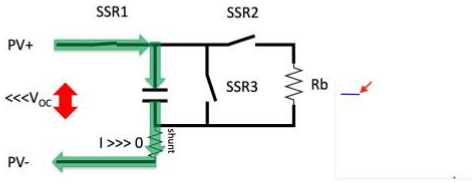
SSR1 fully On - measure I_{SC}

9



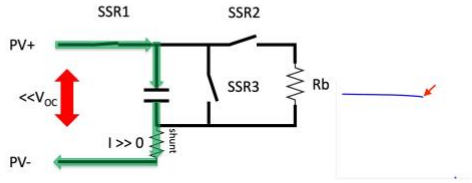
SSR3 Off – caps start charging

10



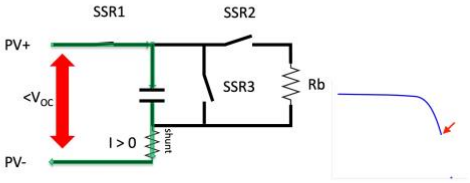
SSR3 Off – caps continue charging

11



SSR3 Off – caps continue charging

12



SSR3 Off – caps continue charging

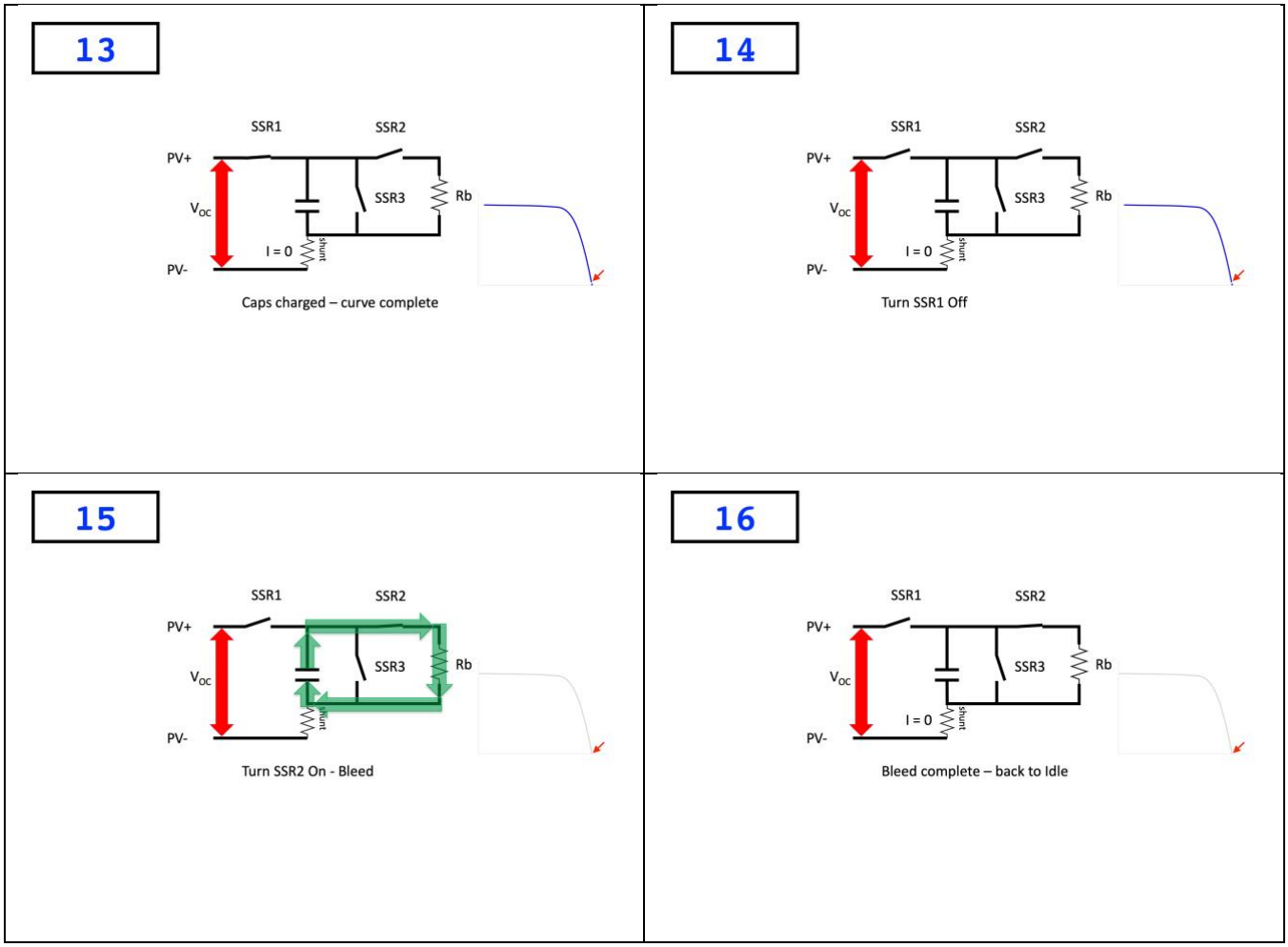


Figure 7-28: SSR Sequencing

7.3.6.1.2 SSR3 Bonus: Advanced Current Calibration

An added benefit of SSR3 is that it enables a much more accurate method of current calibration. In the EMR-based design, it is not possible to simultaneously measure I_{SC} with a multimeter and with the IVS2 ammeter. But by turning both SSR1 and SSR3 On and holding them On for long enough for a human to read the DMM value, this is possible. The software allows current to flow for 3 seconds in this state. It enforces a cool-off period if the current is higher than 6.75 A.

7.3.6.1.3 Arduino SSR control

The SSRs are controlled by the following Arduino pins:

- SSR1 (active-low): pin D2 (same as EMR)
- SSR2 (active-high): pin D6
- SSR3 (active-low): pin D7

No pull-up or pulldown resistors are necessary.

7.3.6.1.4 Documentation in the GitHub Repository

The PCB schematic for the SSR-based PV module version is available in the GitHub repository. There is also a GIF animation that compares it with the EMR-based PV module version schematic, a GIF animation that compares their PCB top layers and another for their bottom layers as well as a GIF that compares their bills of material (BOMs).

7.3.6.2 SSR Version for PV Cells

The EMR version for PV cells uses two EMRs, as described in Section 7.2.4.2 on page 75. Therefore, just to map those two SPDT EMRs to the SPST SSRs requires four SSRs. Then to solve the slow turn-on time problem would require one more, like SSR3 in the PV module version. However, since the PV cell version does not have a bleed resistor, the bleed path can also serve as the load capacitor bypass.

Figure 7-29 below is a diagram of the load circuit path. SSR5 and SSR 6 are the equivalent of the 2nd relay in the EMR version. The equivalent of having the 2nd EMR Off is SSR5 Off and SSR6 On. The equivalent of having the 2nd EMR On is SSR5 On and SSR6 Off. SSR4 activates the bleed path (with SSR1 Off), but also bypasses the load capacitors for the I_{SC} measurement (with SSR1 On).

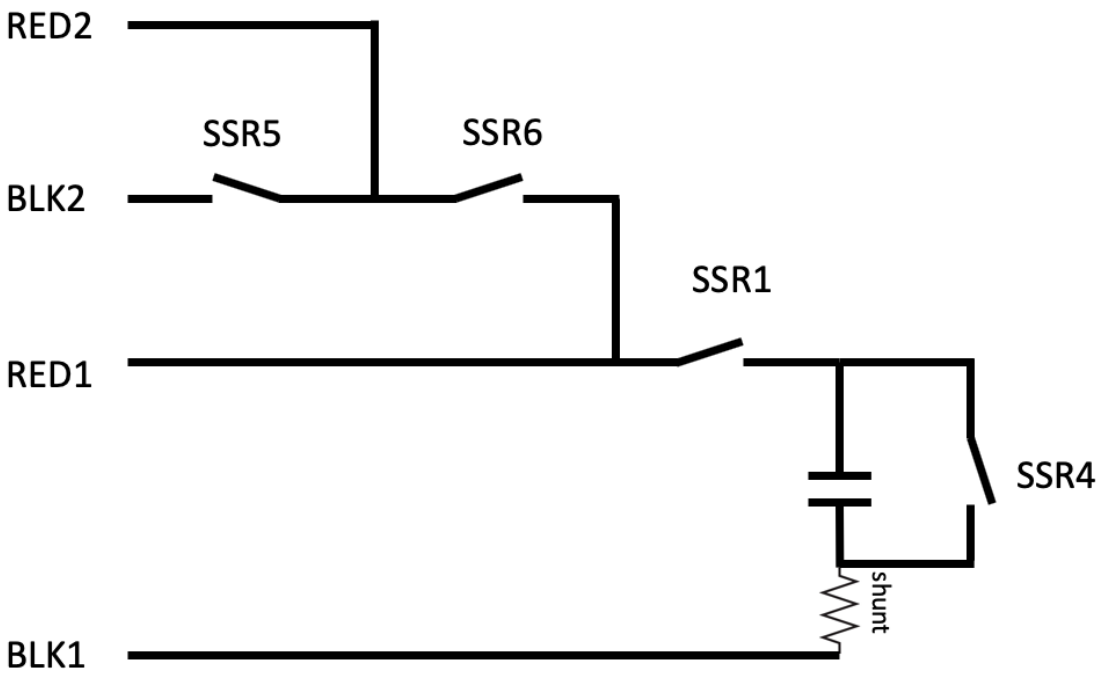


Figure 7-29: SSR-based PV Cell Version Load Circuit

There is no animation for the SSR sequencing of the PV cell version load circuit, but it is pretty straightforward.

SSR5 and SSR6 determine whether the bias battery only, or the series combination of the battery and PV cell are connected to the load circuit. The rest of the load circuit is independent of the states of SSR5 and SSR6.

The idle state is:

SSR1 = OFF
SSR4 = ON

The V_{OC} is measured in this state. The Arduino code then turns SSR1 ON.

Now both SSRs are on. The capacitors are still discharged. There is a path through SSR1 and SSR4 around the capacitors.

Due to the slow turn-on time of SSR1, the voltage does not drop from Voc immediately to 0, and the current does not immediately rise from 0 to I_{SC}. While SSR1 is in the process of turning on, the measurements are following the IV curve backwards. The Arduino code polls the current and voltage measurements, waiting for a stable I_{SC} value (voltage has stopped decreasing and current has stopped increasing). It discards all of the points before detecting the stable I_{SC} condition.

When the stable I_{SC} is detected, SSR4 is turned OFF. Even though SSR4 takes time to turn off, it doesn't affect the IV curve at all. The points that are measured while SSR4 is in the process of turning off are completely valid. It is irrelevant that a small amount of current is going through SSR4 instead of through the load capacitors.

When the curve is complete, SSR1 is turned OFF. Then SSR4 is turned ON to bleed the capacitors. This is again the idle state.

7.3.6.2.1 Arduino SSR control

The SSRs are controlled by the following Arduino pins:

- SSR1 (active-low): pin D2 (same as EMR)
- SSR4 (active-low): pin D8
- SSR5 (active-low): pin D4 (same as 2nd EMR)
- SSR6 (active-low): pin D5

No pull-up or pulldown resistors are necessary.

7.3.6.2.2 Documentation in the GitHub Repository

The PCB schematic for the SSR-based PV cell version is available in the GitHub repository. There is also a GIF animation that compares it with the EMR-based PV cell version schematic, a GIF animation that compares their PCB top layers and another for their bottom layers as well as a GIF that compares their bills of material (BOMs).

7.4 Custom Scaled Versions

IV Swinger 2 was designed for commercial rooftop PV modules. But there is a need for IV curve tracing for much smaller PV modules (and possibly for larger ones). The ranges of the V_{OC} and I_{SC} values for the targeted PV module(s) are the determining factors. This "scaling" can easily be accommodated by using different capacitors and resistors. The same PCBs can be used.

Choosing the appropriate components is the tricky part. It takes some calculations and some of the criteria are subjective. The following sections of this document describe how the values of the standard components were chosen:

- Load capacitors, C1 and C2:
  - Voltage Requirement (Section 3.4.1, page 34)
  - Capacitance Requirement (Section 3.4.2 page 35)
- Bleed resistor, Rb:
  - Resistance (Section 3.5.1, page 43)
  - Power Rating (Section 3.5.2, page 43)
- Voltmeter resistors, R1 and R2:
  - Resistance and power rating (Section 4.2.2, page 49)
- Ammeter resistors, Shunt, Rf and Rg:
  - Resistance and power rating (Section 4.2.3, page 51)

The standard values were chosen to accommodate a wide range of standard commercial rooftop PV modules, but the same equations and reasoning can be used to target modules (or even cells) that are outside this range. This can be a time-consuming and error-prone process that requires a thorough understanding of the above sections of the document. To make it much easier for anyone wanting to build a custom scaled IV Swinger 2, the software now supports a feature that simulates the IV curve that will result from a given V_{OC} and I_{SC} and a particular set of components. Additionally, the simulator can choose the optimal components for a given maximum expected V_{OC} and I_{SC}. This allows users to experiment and be confident in their choices before building the hardware.

8 Software: Arduino Sketch

There are two components to the IV Swinger 2 software:

- The Arduino sketch
- The host (laptop) application software

The Arduino sketch is written in the C++ language variant used for Arduino. The host application is written in Python and is described in the next chapter.

The Arduino sketch performs the following functions:

- Participates in handshakes with the host computer (via USB)
- Receives and processes configuration messages from the host
- Communicates debug messages to the host
- Controls the relay (or the equivalent SSRs) that switch the capacitor between the bleed circuit and the PV circuit
- Reads and records values from the two ADC channels
- Waits for current to stabilize at the beginning
- Compensates for the fact that time passes between the current and voltage measurements
- Selectively discards values so that the Arduino memory isn't exhausted before the IV curve is complete
- Determines when the IV curve is complete
- Reads temperature(s) from optional DS18B20 sensor(s)
- Reads irradiance from the optional ADS1115-based pyranometer
- Uses the ATmega328 internal ADC to measure the 1.1V bandgap voltage relative to Vref
- Sends results to the host

8.1 Compatibility Goals

8.1.1 Hardware Compatibility

The hardware variants were designed such that it is possible for the same Arduino sketch to work on all types of IV Swinger 2 with no modifications or build options. Some parts of the code are only relevant to a particular variant, but they are benign on the other variants. For example, code that activates an SSR is executed even if that SSR does not exist, but since the pin that controls it is not connected to anything in that case, the code has no effect on that variant.

This strategy eliminates the possibility that users might load the wrong code for their variant. The only downside is that the code is a bit more difficult to understand.

8.1.2 Host Application Compatibility

The goal is that any change made to the Arduino sketch be backward compatible with older versions of the host application software, and for any change made to the host application software to be backward compatible with older versions of the Arduino sketch. Of course, new features that require changes in both are not available unless both are updated, but the existing functionality should not break.

It should be noted that this is not tested for every release combination. However, today it is still possible to load the oldest released version of the Arduino sketch and it works with the most recent application release.

The host application software checks the version number of the Arduino sketch when a feature requiring a given sketch level or higher is required and it generates an error dialog if that requirement is not met.

8.2 Host Communication and Handshakes

The sketch communicates with the host via the USB port using the Arduino Serial communication functions. The "host" generally means the application software running on the host laptop, but it is also possible to use the Serial Monitor feature of the Arduino IDE application to manually communicate with the Arduino sketch. This is useful for development testing and for debugging.

The Arduino code uses Serial.print and Serial.println functions to send messages to the host, and for this reason, we'll say that the Arduino code "prints a message". It uses the Serial.available and Serial.read functions to receive messages from the host. All messages are plain ASCII text strings that end with a newline character. Serial.begin is used to set the baud rate to 57600.

There are two types of message sent to the host by the Arduino sketch: solicited and unsolicited. Solicited messages are either explicitly or implicitly requested by the host, i.e. the host waits for the message before its next move. Unsolicited messages are those that the host has not requested, such as informational or debug messages.

The Arduino sketch accepts two types of message from the host: handshake and configuration, both of which are solicited. When the Arduino sketch receives a message from the host, it always echoes back "Received host message: <*msg*>", where <*msg*> is exactly what it received. This echo message is always printed regardless of whether the incoming message is valid.

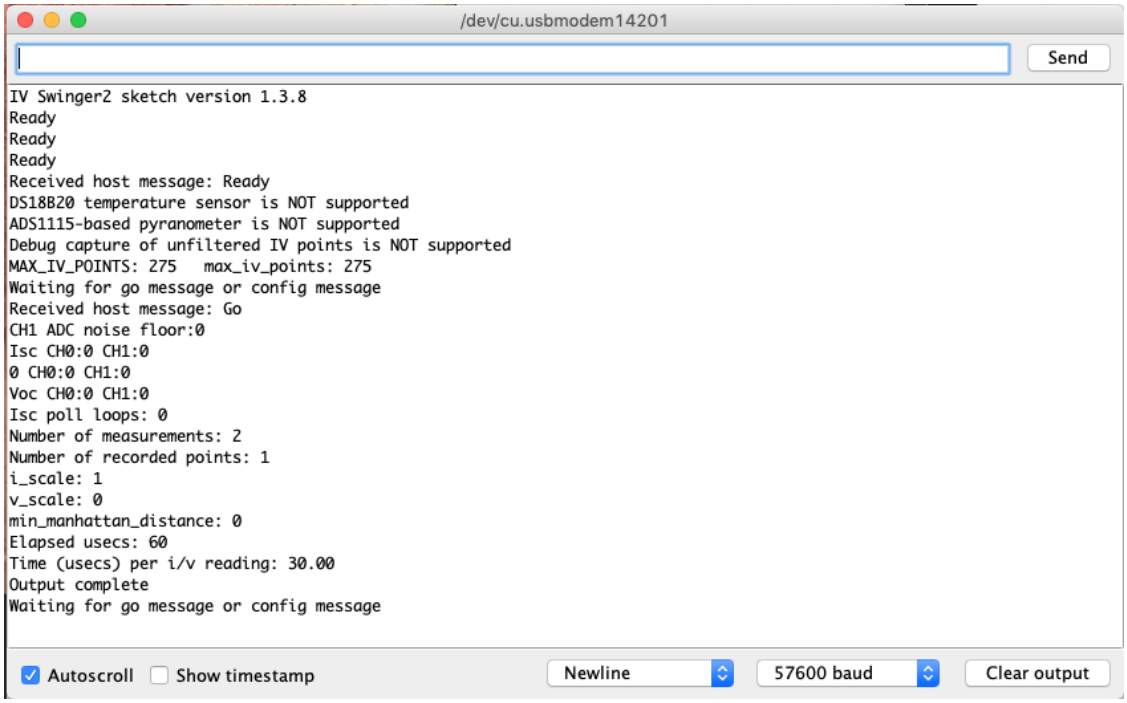
8.2.1 Basic Handshake

The Arduino sketch always begins by printing a message with the sketch version number and then printing a "Ready" message. Here's what that looks like from the Serial Monitor:

At this point, the Arduino code waits for an incoming message from the host. It checks 1000 times, and if no message is received, it prints another “Ready” message and checks 1000 times again. This repeats forever until a message is received. There is a 1 ms delay between each of the 1000 checks, so the “Ready” message is printed about once per second while waiting for the host.

There are two types of message accepted from the host at this point: a “Ready” message or a “Config” message (see the next section). Any other message is just echoed back to the host. A “Ready” message is any message that starts with those five characters (R-e-a-d-y, case-sensitive). When the “Ready” message is received, the sketch prints some informational messages and then prints “Waiting for go message or config message”.

The two types of message accepted from the host at this point are: a “Go” message or a “Config” message (again, see the next section). Any other message is just echoed back to the host. A “Go” message is any message that starts with those two characters (G-o, case-sensitive). When the “Go” message is received, the sketch swings the IV curve and prints the results and other information. It then prints “Waiting for go message or config message” again and is ready to repeat the process.



The above example shows what it looks like when there is nothing connected to the binding posts. Note that no error message is printed in this case; the host is responsible for interpreting the results.
Figure 8-1 below shows a diagram of this basic handshake:

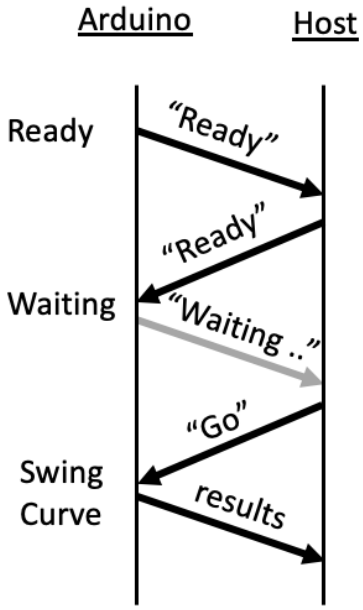


Figure 8-1: Basic Handshake

Note that the gray arrow isn't technically a handshake message because the host is not required to wait for it before sending the "Go" message.

8.2.2 Config Messages from Host

As noted in the previous section, the Arduino code also accepts "Config" messages both when it is waiting for the host's "Ready" message and when it is waiting for the host's "Go" message (not just the first "Go" message, but every time). For the most part, "Config" messages are what they sound like; they provide a way for the host to override defaults in the Arduino code. Originally, that is all that they were. But when the need arose for some other host-to-Arduino commands, it was easier to overload the "Config" message type than to add another type or to rename it, so "Config" is a misnomer for those cases (gray entries in Table 8-1 below).

A "Config" message starts with those five characters (C-o-n-f-i-g, case-sensitive). Any subsequent characters up to the first space character(s) are ignored. The first argument consists of the characters between the first space character(s) and the second space character(s) (or newline). The second argument consists of the characters between the second space character(s) and the newline. Most "Config" messages have one argument. Some have none, and only one (so far) has two. Table 8-1 below lists all of the currently supported "Config" messages.

Config Name	Arg1	Arg2	Description
CLK_DIV	SPI clock divider value	-	Sets *clk_div*
MAX_IV_POINTS	Max IV points value	-	Sets *max_iv_points*
MIN_ISC_ADC	Min Isc ADC value	-	Sets *min_isc_adc*

MAX_ISC_POLL	Max Isc poll value	-	Sets *max_isc_poll*
ISC_STABLE_ADC	Isc stable ADC value	-	Sets *isc_stable_adc*
MAX_DISCARDS	Max discards value	-	Sets *max_discards*
ASPECT_HEIGHT	Aspect height value	-	Sets *aspect_height*
ASPECT_WIDTH	Aspect width value	-	Sets *aspect_width*
WRITE_EEPROM	EEPROM address	Value	Writes value to EEPROM at address
DUMP_EEPROM	-	-	Prints all valid EEPROM address/values
RELAY_STATE	Value (0 or 1)	-	Turns relay or SSR1 off (0) or on (1)
SECOND_RELAY_STATE	Value (0 or 1)	-	Turns 2nd relay off (0) or on (1) Turns SSR5 off/on and SSR6 on/off
DO_SSR_CURR_CAL	-	-	Turns SSR1, SSR3 and SSR4 on for 3 seconds and prints Channel 1 (current) ADC value
READ_BANDGAP	-	-	Reads the ATmega328 internal 1.1V bandgap voltage (relative to Vref) using its internal ADC and prints the total ADC value and iterations

Table 8-1: Config Messages

Figure 8-2 below shows examples of "Config" messages being received at both the places in the handshake where they are accepted. While the sketch is still printing "Ready" once per second, the host sends "Config: DUMP_EEPROM" (the colon is optional, and ignored). In response, the sketch reads all of the valid EEPROM locations and prints their values. Then the "Ready" polling continues. The host does send "Ready" and the sketch waits for "Go". At this point, the host sends "Config: WRITE_EEPROM 8 150001", which causes the sketch to write that value to that address. Then the host sends another "Config: DUMP_EEPROM". The changed value at address 8 can be seen. Finally, the host sends the "Go" command.

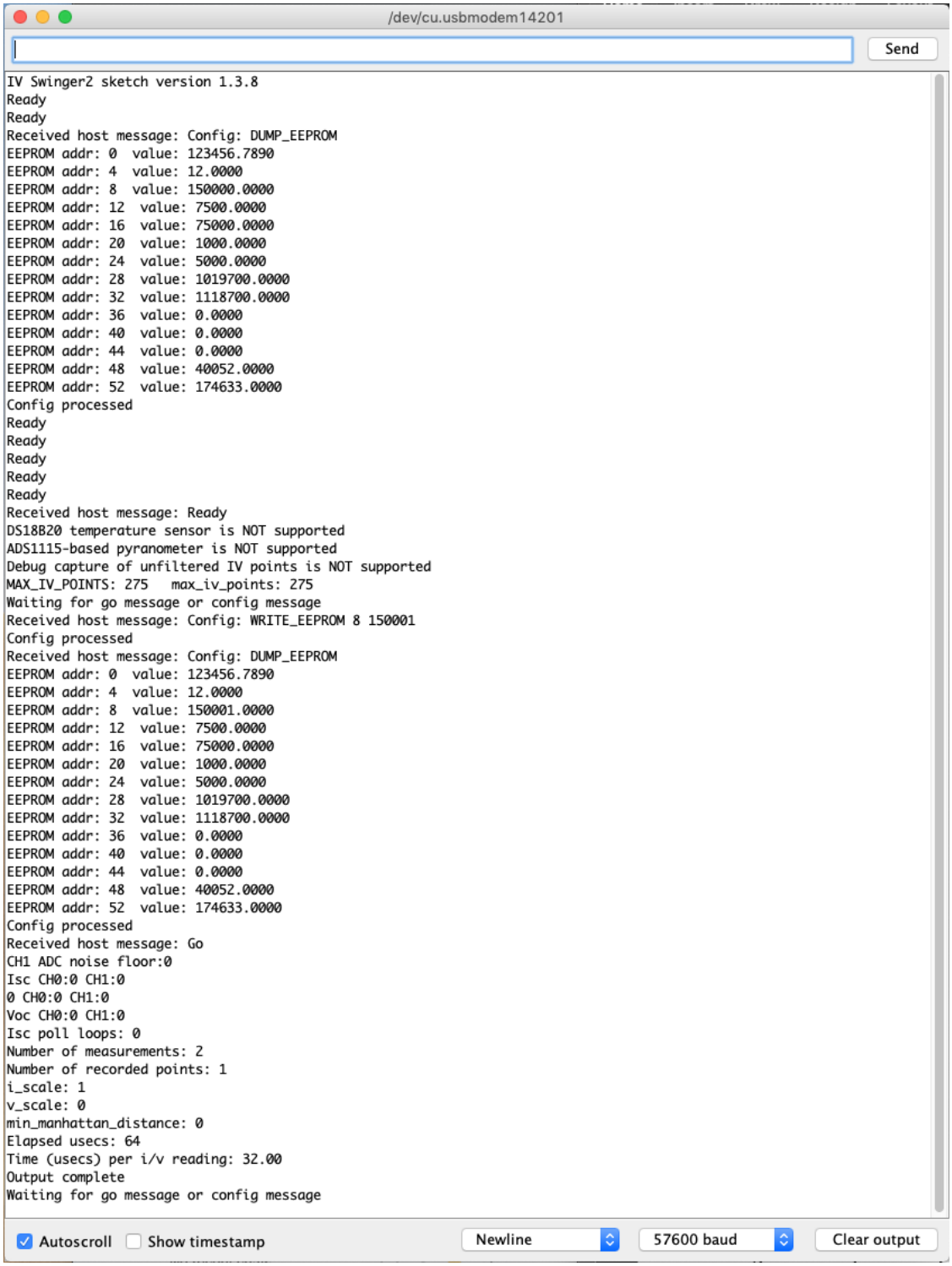


Figure 8-2: Config Message Examples

Note that this example was generated from the Serial Monitor with the host commands being generated by typing them at the top and clicking on the “Send” button. This is not the actual sequence that the host application generates.

8.3 Memory Resources

8.3.1 EEPROM

A very useful feature of the ATmega328 microcontroller used for the Arduino is the EEPROM. This is non-volatile memory, i.e. it retains its value without power. In this sense it is like the flash memory used for the sketch (which is also a type of EEPROM), but it is separate and can be written and read by the sketch.

As shown in the example in Figure 8-2 above, the sketch supports writing values to EEPROM and dumping out all of its valid values.

All values stored in EEPROM by the sketch are 4-byte floating-point numbers.

With a few exceptions, the Arduino sketch is not aware of the meaning of the values that are stored in EEPROM. They are primarily used by the host to save calibration values that are valid for a particular IV Swinger 2. By storing these values on the hardware itself, the calibration "follows" the hardware even if it is used with different laptops. However, the Arduino code doesn't know what the host application is using the EEPROM for and it just takes care of the mechanics of writing to it and reading from it.

The Arduino code does know the meaning of the values of the first two EEPROM locations:

```
EEPROM addr: 0  value: 123456.7890
EEPROM addr: 4  value: 12.0000
```

Address 0 is a "magic number" value that simply indicates that the EEPROM contents are valid. If the EEPROM has never been written, it may contain random values. It is highly unlikely that a random value will match this particular value, so it is a reliable indication that the EEPROM is valid. The value at address 4 is the number of subsequent locations that are valid. When a DUMP_EEPROM command is received, the sketch reads these two locations first. It only dumps if the magic number is found at address 0, and then it uses the value at address 4 to determine how many locations to dump.

The only other location that the sketch knows about is address 44, which (if valid) contains either the value 0 or the value 1, which indicate that the relay is active-low or active-high, respectively.

The meanings of the values at all other EEPROM locations are transparent to the Arduino sketch.

8.3.2 SRAM

The Arduino's 2KB of SRAM must be used judiciously. Since the sketch must store the 12-bit ADC values for both the voltage and current for each recorded point, the number of points that may be recorded is decreased any time SRAM is used for anything else. Each recorded point uses a total of 4 bytes (2 for current and two for voltage), so it would be possible to record 512 points if no SRAM were used for anything else at all.

In addition to minimizing usage of SRAM for anything other than the recorded ADC values, it is also important not to record points that are duplicates or so close to their neighbors that they provide no added value. This will be discussed in Section 8.5.2.5 on page 105.

As described on the Arduino memory tutorial page, strings are a memory hog. Fortunately, it is possible to store static strings in flash (program) memory instead of SRAM, as described in the PROGMEM documentation. The IVS2 sketch uses this religiously (including the F() macro described on that page).

8.3.2.1 ADC Value Arrays

The arrays used for recording all the points of the IV curve except for the I_{SC} and V_{OC} points are named *adc_ch0_vals* and *adc_ch1_vals* and are defined in the *loop()* function. The size of these arrays is determined by the C preprocessor, and depends on whether optional features are supported. If none of the optional features is supported, the ADC value arrays have 275 entries each (FULL_MAX_IV_POINTS). However, if the DS18B20 temperature sensor is supported, this number is reduced by 11 (DS18B20_SRAM/4) and if the pyranometer is supported, the number is reduced by 56 (ADS1115_SRAM/4). There is also a debug option (CAPTURE_UNFILTERED) that reduces it by an amount that depends on how many unfiltered points may be captured.

8.4 Performance

Sketch performance is important. The rate that the curve is "swung" is a function of the capacitor value and the PV module; there is no way to slow it down (other than using a larger capacitance). The faster the software can take measurements, the closer together the points will be, which improves the resolution of the IV curve. Section 3.4.2 on page 35 describes how the part of the IV curve just past the knee is the most performance critical. It also discusses the importance of resolution at the inflection points on shading cases.

Printing messages at any time when the load capacitors are charging is completely forbidden. Even one short message could cause the whole curve to be missed. This is why the values of the points must be stored in arrays in SRAM and printed after the curve is complete.

The ATmega328 microcontroller is a 16-bit processor, so performance is best for operations using 16-bit integer math. Any use of floating-point math, or even 32-bit (long) math slows things down dramatically, so it is not used for any of the code that runs while the capacitor is charging.

A primary performance-limiting factor is the time it takes to read a value from one of the ADC channels over the SPI bus. “Overclocking” the SPI interface speeds things up (and appears to be reliable), but the sketch is still written to minimize the number of ADC reads.

8.5 Arduino *setup()* and *loop()* Functions

Like all standard Arduino sketches, the IV Swinger 2 sketch has *setup()* and *loop()* functions. After a reset[12], *setup()* runs once, and *loop()* runs repeatedly after that.

[12] Things that cause a reset: pushing the reset button, opening the Serial Monitor, connecting the USB cable, host software opening serial port.

8.5.1 setup()

The *setup()* function does the following:

- Gets the relay type (active-low or active-high) from EEPROM location 44
- Initializes the digital output pins
- Initializes the SPI interface
- Sets up the microcontroller's internal ADC to measure the bandgap voltage relative to Vref
- (Optionally) discovers how many DS18B20 temperature sensors are connected and sets their resolution to 10 bits.
- (Optionally) initializes the ADS1115 used in the pyranometer
- Prints the sketch version number
- Prints the "Ready" message to the host
- Waits for "Ready" or "Config" messages from the host
- Processes any "Config" message(s) and continues waiting for "Ready"
- When "Ready" is received, prints informational messages and exits

8.5.2 loop()

The *loop()* function begins immediately after *setup()* exits. It does the following, repeating when it completes.

- Waits for "Go" or "Config" messages from the host
- Processes any "Config" message(s) and continues waiting for "Go"

When "Go" is received:

- Calls the *read_bandgap* function
- Measures V_{OC}
- Measures ADC Channel 1 noise floor
- Turns SSR3 on, turns relay / SSR1 on, turns SSR2 off
- Polls for stable I_{SC}
- Calculates discard criterion (minimum Manhattan distance)
- Loops, measuring and recording I and V points:
    - Checks for "done" (current < threshold)
    - Discards point if it is too close to predecessor
- Turns relay / SSR1 off, turns SSR2 on, turns SSR4 on
- (Optionally) reads pyranometer value
- (Optionally) reads DS18B20 temperature sensor value(s)
- Reports all results to host

Note that the Arduino sketch doesn't know if it is running on an EMR-based or SSR-based IVS2. The steps in which an SSR is turned on or off are executed on an EMR-based IVS2, but have no effect because they are controlling an Arduino pin that is not connected to anything. SSR1 is connected to the same pin as the EMR.

More detail on some of the above steps is contained in the following sections.

8.5.2.1 V_{OC} Measurement

When the "Go" message is received from the host, the relay or SSRs are in the state in which the load circuit is open. It is at this point that the V_{OC} is measured[13].

In early versions of the sketch, the V_{OC} measurement was taken only once. However, it was observed that there often was a variation of several ADC units from one curve to the next, even when the actual V_{OC} was completely stable. This is to be expected, due to minor noise in the circuitry. However, unlike the other points on the curve, the V_{OC} point is not included in the noise reduction algorithm implemented in the host application software. The result was that consecutive IV curves that should have been identical, were slightly different; there was a "jitter" when they were displayed in sequence. The error was quantitatively insignificant but was annoying.

In the current sketch, the V_{OC} ADC value is read 400 times. A frequency count is incremented for each unique ADC value that is read. The ADC value with the highest frequency count (i.e. the mode) is chosen as the V_{OC} value. This results in much more consistent V_{OC} values. Note that since the load capacitors are not charging at this point, the time that it takes to perform this loop (~1/20 second) is not important.

If the V_{OC} ADC value is less than 10 (MIN_VOC_ADC), it is assumed that the IVS2 is not connected to a PV module. The relay (or SSR) is not activated, and the remaining steps are skipped. The V_{OC} value reported to the host is forced to zero.

8.5.2.2 ADC Channel 1 Noise Floor Measurement

While the circuit is open, there is no current flowing. The ADC value on Channel 1 should be 0. However, due to minor noise in the circuitry (and perhaps "not quite" rail-to-rail op amp performance), the actual value read on Channel 1 is often not 0 when the circuit is open.

It is useful to determine what this "noise floor" is for the following reasons:

- To adjust the minimum I_{SC} ADC value
- To determine the Channel 1 ADC value that indicates "done"
- To indicate possible hardware problems

The minimum I_{SC} ADC value is the value on Channel 1 that is the smallest for which an IV curve will be generated. The "done" value is how low the current must fall before the curve is considered complete. Typically, the noise floor is a very small number such as 0, 1 or 2. However, some hardware issues can cause the value to be much higher. The sketch sends the host a log message with the noise floor value, but it does not check it. The host application could (but currently does not) check the value against a threshold and issue a warning it if is too high.

The ADC noise floor is measured in the same loop as the V_{OC} polling. For every read of Channel 0, Channel 1 is also read. The sketch captures the smallest ADC value read, and also the largest. The smallest value is used as the noise floor value. Both the smallest and largest are logged.

[13] Just to be clear, the Arduino sketch simply reads 12-bit values from the ADC. It does not translate the values on Channel 0 to volts, nor does it translate the values on Channel 1 to amps.

8.5.2.3 Activating the EMR or Activating/Deactivating SSRs

Once the VOC ADC value and the ADC Channel 1 noise floor have been determined, the sketch does the following:

- Activates SSR3
- Waits 20 ms
- Activates EMR / SSR1
- Deactivates SSR2

If the hardware is an EMR-based (module or cell) version, that equates to:

- Activates EMR

If the hardware is an SSR-based PV module version, that equates to:

- Activates SSR3
- Waits 20 ms
- Activates SSR1
- Deactivates SSR2

If the hardware is an SSR-based PV cell version, that equates to:

- Activates SSR1

In all cases, this is the state needed to measure I_{SC}. **Note however**, that the EMR-based variants begin charging the load capacitors in this state, but the SSR-based variants are configured to bypass the load capacitors.

8.5.2.4 Polling for a Stable I_{SC} value

Both EMR-based and SSR-based designs take some amount of time to stabilize at a point where the voltage is at its minimum and the current is essentially I_{SC}. How they get there is quite different, but fortunately the same algorithm can be used to poll for when that condition has been reached.

In both cases, the initial few measurements are before the EMR or SSR1 has switched or even started switching. Although the EMR control pin has been activated, the physical relay arm (with the C contact) takes time to move from the NC contact to the NO contact, so the first few measurements are still of the open circuit condition. The SSR has just started its very slow turn on, so it also looks nearly open-circuit.

In the EMR case, once the C and NO contacts touch, current begins flowing quite rapidly and voltage drops to nearly zero equally rapidly. Due to parasitic inductance, current cannot go from zero to I_{SC} immediately, however. There are usually some points measured that are before the current has risen to I_{SC}. There also can be a short period of overshoot and "ringing" before the current stabilizes. There also can be a "bounce", where the contacts lose connection momentarily before making solid contact again.

In the SSR case, the slow turn-on time of SSR1 results in the measured points essentially moving along the IV curve backwards (from V_{OC} to I_{SC}) as described in Section 7.3.6.1.1 on page 87. Because SSR3 is bypassing the load capacitors, the current and voltage both stabilize at values that are as close to the I_{SC} point as possible.

The stable I_{SC} polling uses a sliding window of three (I,V) measurements. Each time through the loop, it discards the oldest point and reads the (I,V) values for a new point. This continues until the following conditions are met:

- The ADC Channel 1 (current) values of the three points are greater than the configured minimum I_{SC} ADC value
- The ADC Channel 0 (voltage) values of the three points are increasing or equal
- The ADC Channel 1 (current) values of the three points are decreasing or equal
- The ADC Channel 1 (current) value differences between consecutive points is less than or equal to the configured I_{SC} stable ADC value

When all of those conditions are met, the code breaks out of the loop. The ADC Channel 1 (current) value of the oldest of the three points is considered to be the I_{SC} value. The newest of the three points is saved as the first recorded point (location 0 in the ADC value arrays). The middle point is not used.

If the stable I_{SC} conditions are not met in the configured maximum number of I_{SC} polling loops, a polling timeout is flagged and a message to that effect is sent to the host. Otherwise, the discard criterion is calculated and the remaining IV curve points are captured as described below in Sections 8.5.2.5 and 8.5.2.6

The SSR-based designs need to have their bypass SSR (module:SSR3, cell:SSR4) turned off in order to start the load capacitors charging, so the rest of the IV curve can be traced. This is done inside the stable I_{SC} polling loop when it is detected that all three of the points in the sliding window have the same ADC Channel 0 (voltage) value. That indicates that SSR1 has fully turned on. After deactivating the Arduino pin controlling SSR3/SSR4 (which initiates the turn-off), the ADC Channel 0 (voltage) value is monitored until it has increased by 10 ADC units. At that point, the stable I_{SC} polling is restarted. All of this is done without even knowing if the hardware is SSR-based. However, an EMR-based design is not likely to ever hit the condition where three consecutive points have the same ADC Channel 0 (voltage) value.

Unlike the V_{OC} polling, the polling for stable I_{SC} is very performance-sensitive because the load capacitors are starting to charge (in the EMR-based designs). In addition to using only 16-bit integer variables, the code that is used to check for the stable I_{SC} conditions is written as nested "if" statements rather than a long series of conditions connected by "&&". Depending on how good the Arduino compiler is, this may or may not make a difference. It's probably insignificant relative to the ADC reads, but can't hurt.

8.5.2.5 Calculating the Discard Criterion

The downside of taking measurements quickly is that too many measurements are taken during the parts of the curve where the sweep rate is low. If all these points are recorded, SRAM will be exhausted before the curve is complete. The software must selectively discard points to prevent this from happening. The trick is to determine which points to discard. It is not useful to have points that are very close to each other, so the discard criterion is based on the distance between points. This calculation has to be very fast because it is performed after every measurement, and that reduces the rate that measurements can be taken. Any use of floating-point math, or even 32-bit (long) math slows things down dramatically, so only 16-bit integer math is used. Instead of Pythagorean distance ($\sqrt{\Delta x^2 + \Delta y^2}$), so-called Manhattan distance ($\Delta x + \Delta y$) is used, which is faster to calculate.

There are two goals to the discard algorithm:

1. Avoid running out of memory (crucial)
2. Achieve visually uniform point spacing (aesthetic)

The minimum distance criterion calculation requires some computation time after the stable I_{SC} polling and before capturing the remaining IV curve points, but that is not normally a resolution-sensitive part of the curve. Nevertheless, this code is also restricted to simple 16-bit integer math in order to make it as quick as possible.

A rudimentary Manhattan distance between two (I,V) points could simply be defined as the sum of the ADC differences on the two channels. However, that would not account for the fact that the two dimensions may have very different scales. For example, if the V_{OC} ADC value is 1300 and the I_{SC} ADC value is 3100, the visual distance on the graph of 10 ADC units is much larger in the voltage (horizontal) dimension than 10 ADC units in the current (vertical) dimension. If the unscaled ADC values are used, a discard criterion of 10 ADC units would result in points that are further apart on the horizontal part(s) of the curve, and closer together on the vertical part(s) of the curve. Scaling the ADC values results in a much more uniform point spacing, which is nicer-looking.

The function *compute_v_and_i_scale()* determines the integer scaling values for the V (channel 0) and I (channel 1) ADC values. The two computed values must add up to 16 or less. This is so that when they are multiplied by the 12-bit ADC values and added, the result cannot exceed the maximum 16-bit unsigned integer value. The computed *v_scale* and *i_scale* values are used to calculate the minimum Manhattan distance, and are also used to calculate the Manhattan distance between each measured point and the previous recorded point to which the minimum is compared. The function is run only once because it is based only on the V_{OC} and I_{SC} ADC values, which are constant. Since the graphs are rendered in a non-square rectangular aspect ratio, the scales of the axes differ. The initial scaling values could be:

```
initial_v_scale = aspect_width / voc_adc
initial_i_scale = aspect_height / isc_adc
```

That would require large values for aspect_width and aspect_height to use integer math. Instead, proportional (but much larger) values are computed with:

```
initial_v_scale = aspect_width * isc_adc
initial_i_scale = aspect_height * voc_adc
```

Using the example from earlier, if the V_{OC} ADC value is 1300 and the I_{SC} ADC value is 3100, and aspect_width is 3 and aspect_height is 2 (the defaults):

```
initial_v_scale = 3 * 3100 = 9300
initial_i_scale = 2 * 1300 = 2600
```

An algorithm is then performed to reduce the values proportionally such that the sum of the integer values is 16 or less. Of course, there is some rounding error. For this example:

v_scale = 9
i_scale = 3

The *v_scale* and *i_scale* values produced by the *compute_v_and_i_scale()* function can now be used to calculate the Manhattan distance between any two points:

Equation 8-1: Manhattan Distance

$$\text{Manhattan distance} = \text{ADC channel 0 delta} * v\_scale + \text{ADC channel 1 delta} * i\_scale$$

That needs to be compared with a minimum Manhattan distance, which can be calculated once, just after the *compute_v_and_i_scale()* function has determined the *v_scale* and *i_scale* values. The worst-case length of an IV curve is the Manhattan distance between its I_{SC} point and its V_{OC} point. We also know how many points (N) we can capture without running out of memory. Therefore:

Equation 8-2: Minimum Manhattan Distance

$$\text{Minimum Manhattan distance} = \frac{\text{Manhattan distance}(I_{SC}, V_{OC})}{N}$$

The number N is nominally the number of entries in the arrays that record the ADC values for each channel (MAX_IV_POINTS). However, the user may choose to configure a smaller value. This will usually result in a number of captured points that is a fair amount lower than N. The N value is how many points there would be if all points were the minimum distance apart, and the actual distance between the I_{SC} point and the V_{OC} point were equal to the Manhattan distance. But some points will be farther apart than the minimum distance. One reason is simply because, unless N is set to a very small number, there are portions of the curve where the limiting factor is the rate that the measurements can be taken; even without discarding measurements, the points are farther apart than the minimum. The other reason is that it is unlikely that a measurement comes at exactly the minimum distance from the previously recorded measurement, so the first one that does satisfy the requirement may have overshot the minimum by nearly a factor of 2:1 in the worst case. And, of course, the actual IV curve is always shorter than the Manhattan distance.

8.5.2.6 Capturing the Remaining IV Curve Points

Once minimum Manhattan distance has been calculated, the sketch proceeds to read the remaining points on the IV curve and add their ADC values to the ADC value arrays.

8.5.2.6.1 IV Skew Compensation

A single point on the curve requires reading both channels of the ADC. There is no way to read both values at the same time. Each read requires a separate SPI transaction, so some time passes between the two reads, and the values do not represent the exact same point in time. On diagonal parts of the curve, this results in a skew, as illustrated in Figure 8-3 below.

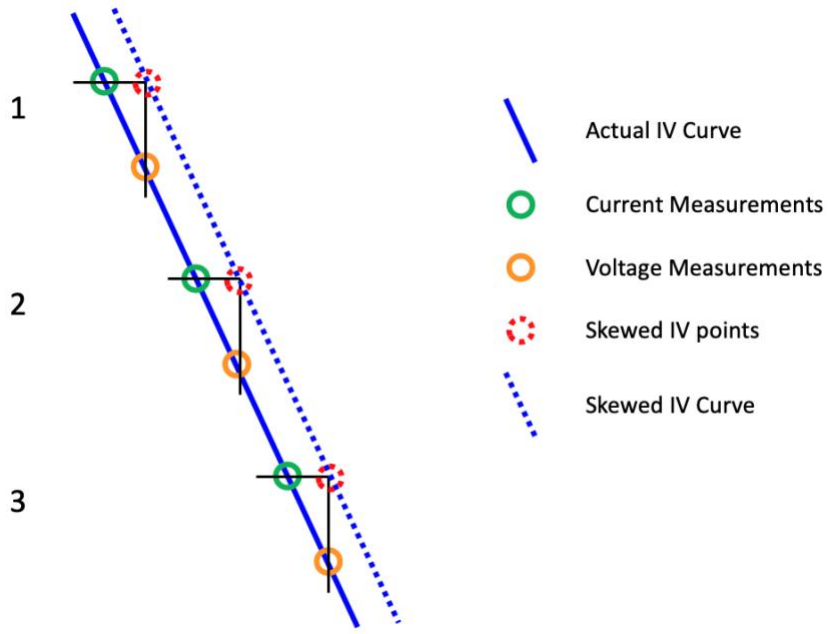


Figure 8-3: IV Curve Skew Due to Time Passage Between I and V Measurements

The simplest way to deal with this would be to ignore it; if the points are close enough together, the effect is relatively minor. But it isn't difficult to compensate for, so we do. One way to compensate would be to do three reads for each point (i.e. Channel 0, Channel 1, Channel 0 or Channel 1, Channel 0, Channel 1) and average the first and third. But that would slow things down by 50%. Instead, we just do one read of each channel on each iteration, but interpolate between the Channel 1 (current) values of each iteration. The catch is that there is computation between iterations (which takes time), so it's not a simple average; it's a weighted average based on measured times.

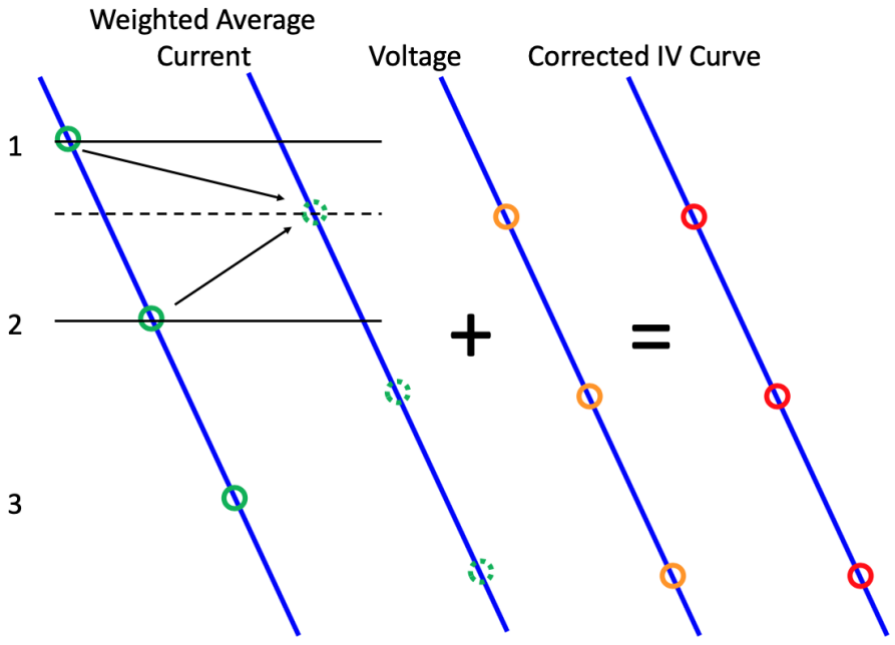


Figure 8-4: Using Weighted Average to Infer Current at Time of Voltage Measurement

Figure 8-4 above shows conceptually how this works. The actual order of events is:

- ADC Channel 1 (current) value #1 is read and recorded
- ADC Channel 0 (voltage) value #1 is read and recorded
- ADC Channel 1 (current) value #2 is read and recorded
- Recorded ADC Channel 1 (current) value #1 is replaced by weighted average of #1 and #2 values

This results in a recorded ADC Channel 1 (current) value #1 that is approximately the value that it would have been if it had been possible to read it simultaneously with the ADC Channel 0 (voltage) value #1.

8.5.2.6.2 Done Check

Every ADC Channel 1 (current) value that is read is checked to see if the tail of the curve has been reached. Due to the charging characteristics of the load capacitors, the progress toward the V_{OC} point at this end of the curve continues to slow down the closer it gets. In fact, it just keeps getting closer, but would never actually reach the V_{OC} point. Since the V_{OC} point was measured with the circuit open, there is a point of diminishing returns where waiting longer is pointless.

The "done" value for the ADC Channel 1 (current) value is determined before the load circuit is closed. It is twice the noise floor or 20 ADC units, whichever is less. Every Channel 1 read is checked against this "done" value. If it is less than the "done" value AND the delta between it and the previous Channel 1 value is less than 3, then the curve is considered complete, and no more measurements are taken.

8.5.2.6.3 EMR Contact Bounce Handling

Each point on the IV curve *should* have an ADC Channel 0 (voltage) value greater than or equal to its predecessor's. However, when the EMR is activated, the C contact can "bounce" off the NO contact, meaning the connection is made but then momentarily lost (or partially lost) before making solid contact again. This usually will occur during, and be filtered out by, the stable I_{SC} polling (Section 8.5.2.4). But it is possible for three points to satisfy all of the stable I_{SC} polling conditions before the bounce stops.

Figure 8-5 below shows an actual case of EMR contact bounce. It does not show the points prior to the I_{SC} stable points. The first two points are normal, but the third point (in the lower right corner) is the result of the bounce. The contact resistance momentarily looks high (although not an open circuit), causing the voltage to be high and the current to be low. However, all of the criteria for stable I_{SC} are met. The fourth point is back where it should be, and the curve proceeds.

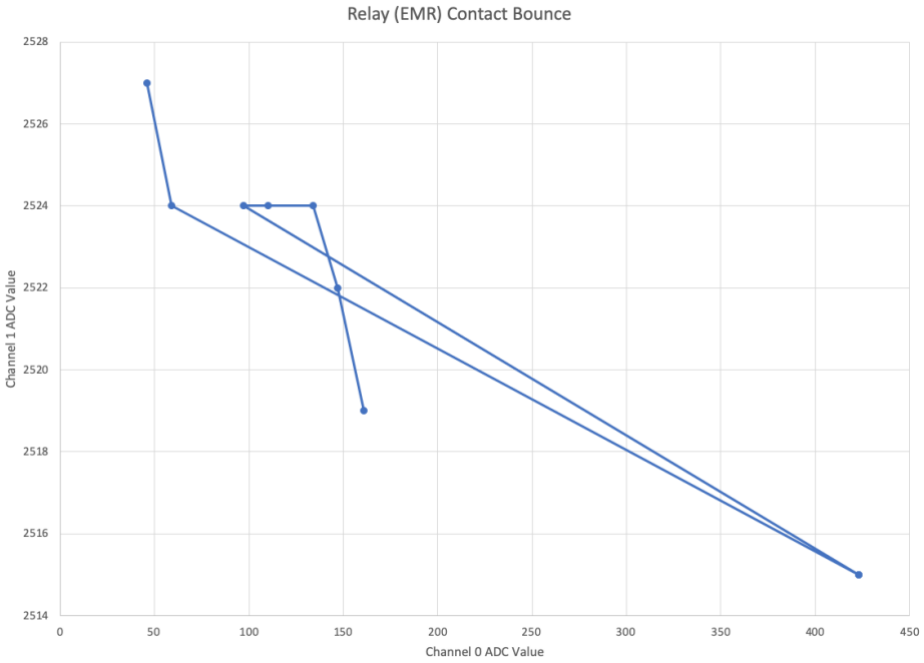


Figure 8-5: EMR Contact Bounce

This backward excursion causes several problems, so the sketch detects this case and removes the erroneous point (or points). It does this by looking at the ADC Channel 0 (voltage) value of each point and checking if it is less than the preceding point's. If so, then it searches backwards through the previous points until it finds one that has a lower voltage and replaces its successor with the current point and rewinds the point number counter.

In the case shown in Figure 8-5, this condition is detected when it processes the 4th point. Its Channel 0 ADC value is 97, which is less than the 423 of the 3rd point. It searches backward until it finds the 2nd point, which has an ADC value of 59, which is less than 97. It then decrements the point counter so that the 4th point is saved as the 3rd point, overwriting the erroneous 3rd point.

While it is probably not possible for the bounce to span more than two or three points, this algorithm covers the general case of it spanning N points (and starting at any point).

8.5.2.6.4 Discard Decision

Section 8.5.2.5 on page 105 described why discarding points is necessary and described the discard criterion, which is a minimum Manhattan distance.

Every point that is measured is checked against the previous non-discarded point to see if its Manhattan distance (Equation 8-1) from that point is greater than or equal to the minimum Manhattan distance (Equation 8-2). If it is not, then the point is discarded. This is done by suppressing the incrementing of the point counter (causing the next point to overwrite it).

The number of discards since the last non-discarded point is incremented on each discard. There is a configured maximum number of discards. If that value is reached, the comparison with the minimum Manhattan distance is overridden, and the point is not discarded. If Max Discards is configured to be zero, then no points are discarded. This can be useful to demonstrate why discarding is necessary.

8.5.2.7 Deactivating the EMR or Deactivating/Activating SSRs

After the last point on the IV curve is measured and recorded, the sketch does the following:

- Deactivates EMR / SSR1
- Activates SSR2
- Activates SSR4

If the hardware is an EMR-based (module or cell) version, that equates to:

- Deactivates EMR

If the hardware is an SSR-based PV module version, that equates to:

- Deactivates SSR1
- Activates SSR2

If the hardware is an SSR-based PV cell version, that equates to:

- Deactivates SSR1
- Activates SSR4

In all cases, this disconnects the PV from the load circuit and bleeds the load capacitors.

8.5.2.8 Reading the Pyranometer Value

If the pyranometer option is enabled (ADS1115_PYRANOMETER_SUPPORTED defined), the ADS1115 channel 2 is read in single-ended mode to get the photodiode temperature from the TMP36 sensor. It does this in a loop, adding up the values that it reads. If it reads a value of -1, there is no ADS1115 present, and if it reads a value less than 4000, there is no TMP36.

If the ADS1115 and TMP36 are both present, the average TMP36 value is calculated. Then the loop is repeated. This time each value is compared with the average. If it differs by 0.5% (MAX_STABLE_TEMP_ERR_PPM) or more, a stable value has not been found and the whole thing is retried, including the loop that is used to determine the average. This algorithm is used because the TMP36 occasionally returns spurious values. If a stable value is not found in 20 tries, a warning message is printed. Otherwise, a message is printed with the average value, e.g.:

```
Arduino: ADS1115 (pyranometer temp sensor) raw value: 10935
```

Next, if the ADS1115 is present, channels 0 and 1 (photodiode) are read in differential mode. The same algorithm that is used for the TMP36 is used to find a stable irradiance value, retrying if necessary. The stability criterion is 1% (MAX_STABLE_IRRAD_ERR_PPM). If a stable value is not found in 20 tries, a warning message is printed. Otherwise, a message is printed with the average value, e.g.:

```
Arduino: ADS1115 (pyranometer photodiode) raw value: 13823
```

The sketch does not convert these raw values to temperature and irradiance and does not perform the temperature compensation. It just sends the raw values to the host and the host application performs those calculations.

8.5.2.9 Reading the DS18B20 Temperature Sensor Value(s)

If the DS18B20 temperature sensor option is enabled (DS18B20_SUPPORTED defined), each of the DS18B20 sensors that was found during *setup()* is read, and its sensor number and temperature value is printed, e.g.:

```
Temperature at sensor #1 is 48.75 degrees Celsius
Temperature at sensor #2 is 48.50 degrees Celsius
```

8.5.2.10 Reporting Results to Host

The ADC Channel 0 (voltage) value and the ADC Channel 1 (current) value of each recorded point are printed, in order. The I_{SC} point is printed first. Next all of the "middle" points are read from the arrays in SRAM and printed. Finally, the V_{OC} point is printed.

The I_{SC} point is not really known because the minimum load circuit resistance is not zero. Its approximation is printed. This is a voltage of 0 and the ADC Channel 1 (current) value of the first of the three consecutive points that satisfied the stable I_{SC} criteria. Sometimes this does not turn out to be a very good approximation. The host application software has more sophisticated algorithms for extrapolating a more accurate I_{SC} value.

After the IV curve points, several messages with debug information about the run are printed. The last thing that is printed is "Output complete".

This is the end of the *loop()* function, which is then called again.

8.6 Utility Functions

8.6.1 bool *get_host_msg*(char * *msg*)

The *get_host_msg* function is called when a handshake or config message is expected from the host. The caller passes it a pointer to a string variable (character array). It reads one serial character at a time using Serial.*read()* when Serial.*available()* indicates that one or more characters are present in the serial input buffer. Each received character is appended to the *msg* string. When a newline (\n) character is received, the message is complete. Since a C string requires a NULL (\0) character at the end, the newline character is replaced by a NULL character at the end of the *msg* string.

A message timeout is implemented in the *get_host_msg* function. The *msg_timer* variable is initialized to 1000 (MSG_TIMER_TIMEOUT). If the call to Serial.*available*() returns a value of zero, *msg_timer* is decremented. If the call to Serial.*available*() returns a non-zero value, *msg_timer* is restored to 1000. There is a 1 ms delay between each time Serial.*available*() is called, so if no characters are received in approximately one second, *msg_timer* will decrement to zero. If this happens, a timeout has occurred, and the function stops checking for received characters.

The number of characters received is also checked. If 35 (MAX_MSG_LEN) characters have been received, and the last one is not a newline character, the message is too long. The function prints an error message and stops checking for received characters.

The function returns:

true: if a message was received successfully
false: if a timeout was detected or if the message is too long

8.6.2 void *process_config_msg*(char * *msg*)

The *process_config_msg* function is called if a message from the host begins with the characters "Config". The caller passes it a pointer to the string variable with the received message (including the "Config"). The message string is parsed into fields, using the space character as the delimiter. The first field after the "Config" field is the *config_type*. The second field (if it exists) is the *config_val*. The third field (if it exists) is the *config_val2*.

The function then compares the *config_type* string with each of the known config messages (see Table 8-1 on page 98) and processes it accordingly. The config messages that are not colored gray in that table simply set the value of a global variable. The one exception is the CLK_DIV message, which also calls the SPI.*setClockDivider()* function, to change the SPI clock frequency.

The config messages listed in gray in Table 8-1 take the action specified in that table. Most call one of the functions described in the following sections. The exception is the WRITE_EEPROM message, which uses the EEPROM.*put()* function to write specified value to the specified address. It also snoops the address, and if it is 44 (EEPROM_RELAY_ACTIVE_HIGH_ADDR), it sets the *relay_active* and *relay_inactive* global variables and sets the relay and 2nd relay control pins to their new inactive value.

8.6.3 void *dump_eeprom*()

The *dump_eeprom* function is called when a DUMP_EEPROM config message is received. It dumps the valid EEPROM entries using EEPROM.get(). The determination of which entries are valid was described earlier.

8.6.4 char *get_relay_active_val*()

The *get_relay_active_val* function is called at the beginning of the *setup()* function. It reads EEPROM location 44 (EEPROM_RELAY_ACTIVE_HIGH_ADDR) and returns either LOW or HIGH. There is code to check that the EEPROM is programmed and that location 44 is valid. If either of those checks fails, a value of LOW is returned, since that is the default.

8.6.5 void *set_relay_state*(bool *active*)

The *set_relay_state* function is called when a RELAY_STATE config message is received. It turns the EMR (or SSR1) on if it is called with *active* = true, and it turns it off if it is called with *active* = false

Note that the host application software doesn't currently use the RELAY_STATE config message for anything, but it can be useful for debugging when typed manually in the Arduino IDE Serial Monitor.

8.6.6 void *set_second_relay_state*(bool *active*)

The *set_second_relay_state* function is called when a SECOND_RELAY_STATE config message is received.

If called with *active* = true, it:

- Turns SSR6 off
- Turns the 2nd EMR or SSR5 on

If called with *active* = false, it:

- Turns the 2nd EMR or SSR5 off
- Turns SSR6 on

This is used for the cell versions when there is a bias battery. Note that the host application is completely in control. The two IV curves are separately initiated by the host application.

8.6.7 void *do_ssr_curr_cal*()

The *do_ssr_curr_cal* function is called when the DO_SSR_CURR_CAL config message is received. The host application generates this config message when the user performs an advanced current calibration and has specified that the hardware is SSR-based. See Section 7.3.6.1.2 on page 90 for a description of this feature.

The function begins by calling the *read_bandgap* function. This allows the host application to determine the Vref voltage.

The function proceeds by:

- Activating SSR3
- Activating SSR4
- Deactivating SSR2
- Activating SSR1

For a PV module variant, this equates to:

- Activating SSR3
- Deactivating SSR2
- Activating SSR1

For a PV cell variant, it equates to:

- Activating SSR4
- Activating SSR1

In both cases, the “PV” (more likely a bench power supply) is connected to the path that bypasses the load capacitors.

Next, the function loops for three seconds (SSR_CAL_USECS). For most of that time, nothing happens in this loop. The purpose of the loop is to allow the current to flow long enough that the user has time to look at the value on the DMM and for it to stabilize. In the last one tenth of a second (SSR_CAL_RD_USECS) of the 3-second loop, the code reads the ADC Channel 1 (current) value on each iteration. It adds up all of the values that it reads, keeps a count of how many values it read, and tracks the minimum and maximum values. After the three seconds have elapsed, the loop exits.

While reading the ADC, if a value of 4095 is read, the following message is printed, and the loop exits:

```
SSR current calibration: ADC saturated
```

If no saturated value was read, the function uses the sum of the ADC values and the read count to calculate the average read value. It then performs a 1% stability check: it multiplies the difference between the minimum and maximum values by 100 and compares that to the average value. If it is larger, a message like the following is printed:

```
SSR current calibration ADC not stable.  Avg: 1376  Min: 1350  Max: 1393
```

The function ends by:

- Deactivating SSR1
- Activating SSR2
- Deactivating SSR3
- Deactivating SSR4

For a PV module IVS2 variant, this equates to:

- Deactivating SSR1
- Activating SSR2
- Deactivating SSR3

For a PV cell IVS2 variant, it equates to:

- Deactivating SSR1
- Deactivating SSR4

If no saturated value was read, and the stability check passed, the function prints a message like the following:

```
SSR current calibration ADC value: 2383
```

The ADC value is the average value.

8.6.8 void *set_up_bandgap*()

The *set_up_bandgap* function sets up the ATmega328 microcontroller's internal 10-bit ADC (not to be confused with the external 12-bit MCP3202 ADC) to measure the bandgap voltage relative to Vref (aka Vcc). Specifically, it programs the internal ADC Multiplexer Select (ADMUX) register as follows:

- REFS[1:0] = 2'b01 : Voltage Reference Selection = AVcc
- ADLAR = 1'b0 : Left Adjust Result = False
- MUX3..0 = 4'b1110 : Input Channel Selection = 1.1V (V_{BG})

The *set_up_bandgap* function is called by the *setup()* function because this programming needs to be done only once since the internal ADC is not used for anything else by the IV Swinger 2 Arduino code.

8.6.9 int *read_internal_adc*()

The *read_internal_adc* function performs a single read of the ATmega328's internal 10-bit ADC. Since the *set_up_bandgap* function has already programmed the ADMUX register, this function just needs to:

- Start the conversion by setting the ADSC bit in the ADCSRA register
- Poll that bit for it to clear
- Read the ADC register and return its value to the caller

8.6.10 void *read_bandgap*(int *iterations*)

The *read_bandgap* function calls the *read_internal_adc* function *iterations* times. It accumulates the total internal ADC value returned across all iterations. It then prints that total internal ADC value and the number of iterations in a message like the following:

```
Bandgap total ADC: 219015 iterations: 1000
```

The *read_bandgap* function is called by the following:

- The *loop* function, each time a "Go" message is received, immediately before swinging the curve
- The *process_config_msg* function, when a READ_BANDGAP config message is received
- At the beginning of the *do_ssr_curr_cal* function

The message sent to the host can be used to calculate the average value of the internal ADC. In the example above, this average value would be 219.015. Since the internal ADC is 10 bits, this means the measured bandgap voltage is 219.015/1024 of the reference voltage (Vref). Depending on which voltage is known (Vref or bandgap), this can be used to calculate the other.

8.6.11 int *read_adc*(int *ch*)

The *read_adc* function is called many places in the sketch. The caller specifies the channel number (0 or 1) and the function uses the SPI.transfer() to read the MCP3202 value on that channel. It returns the requested 12-bit value as a 16-bit integer. Refer to the MCP3202 datasheet for a better understanding of this function. This code was basically copied from somewhere that I can't remember and cleaned up.

8.6.12 void *compute_v_and_i_scale*(int *isc_adc*, int *voc_adc*, int * *v_scale*, int * *i_scale*)

The *compute_v_and_i_scale* function is discussed in Section 8.5.2.5 on page 105.

9 Software: Host Application

The host application software runs on the laptop that is connected to the IV Swinger 2 hardware. It also can run without connecting to the hardware in order to browse and modify previously recorded IV curves.

9.1 Scope

According to cloc, there are over 15k physical lines of code in the seven main Python files. Documenting the details of that much code here is not feasible. This document will cover the high- to mid-level software design. Along with the 15k lines of code are over 8k lines of comments, which should serve to document the low-level details.

9.2 Language

In IV Swinger 2 release 2.7.0, support for Python 3.x was added. Prior to that, the Python code was compatible with Python 2.7.x only. As of this writing, there are two separate, but functionally equivalent codebases: IV_Swinger/python for the Python 2 code and IV_Swinger/python3 for the Python 3 code. At some point, the Python 2 support may be dropped. This document was originally written based on the Python 2 code, but has been updated for Python 3. However, it is quite possible that there are some places that were missed.

9.3 Software Design Objectives

The objectives of the host application software design were:

- Reuse IV Swinger 1 Code
- Keep UI code separate from UI-independent code
- Support both Windows and Mac and laptops

9.3.1 Reuse IV Swinger 1 Code

The main reason for choosing Python was because the original IV Swinger software was written in Python. That choice was more obvious because it ran on a Raspberry Pi, and Python is the most popular and most supported language on that platform.

A reasonable amount of the code in the legacy IV_Swinger.py and IV_Swinger_plotter.py modules is used for IV Swinger 2. This is not copy/paste reuse, but actual importing of those modules. Some changes were made to those modules, but they are still compatible with IV Swinger 1.

9.3.2 Keep UI Code Separate from UI-independent Code

Although the currently supported application provides a standard graphical user interface (GUI), the software is structured so there is a clean separation between the GUI code and the UI-independent code. This makes it possible to implement alternate user interfaces, such as a command-line interface (CLI). It also is possible for a Python script to use the UI-independent code to implement automated testing, potentially also interacting with other devices. Imagine, for example, a device that incrementally moves

"shade" across the PV module - a script could loop, moving the shade a step and swinging an IV curve on each iteration.

9.3.3 Support Both Windows and Mac Laptops

It was important that the application be able to run on most laptops, especially those used by college students. With the ability to run on both Windows and Mac laptops, the vast majority of users are covered. In theory, the app should also run on Linux/Unix, but that has not been tested (and no one has yet requested it).

9.4 GUI Framework Choice

There are many so-called GUI frameworks available for Python. The requirements were:

- Cross-platform (Windows/Mac/Linux)
- Look and feel adapts to platform
- Free / open source
- Well-documented and supported

tkinter/tkinter.ttk satisfied all of those requirements. It is also the only one that is included in the Python distribution itself, which was a big plus. There are advocates of the other choices, of course. Some have "GUI builder" tools that generate the code for you; tkinter/tkinter.ttk does not. One of the biggest criticisms of tkinter was a reputation for looking like XWindows (old Unix window system) on all platforms; it had the reputation of being "ugly". That was fixed years ago with the tkinter.ttk "themed toolkit", however, and now tkinter/tkinter.ttk apps acquire the look and feel of the platform they run on.

9.5 Python Modules

Table 9-1 below shows the Python modules (i.e. files) that are found in the "python3" directory of the GitHub repository and whether they are used for IVS1, IVS2, or both. The IV_Swinger_test.py and all of the Adafruit_*.py modules are used for IVS1 only and are not covered in this document. Tooltip.py and myTkSimpleDialog.py are slightly modified versions of public modules.

Module	IVS1	IVS2
IV_Swinger.py	✔	✔
IV_Swinger_plotter.py	✔	✔
IV_Swinger2.py		✔
IV_Swinger2_gui.py		✔
IV_Swinger2_sim.py		✔
IV_Swinger_PV_model.py		✔
IV_Swinger2_PV_model.py		✔
Tooltip.py		✔
myTkSimpleDialog.py		✔
IV_Swinger_test.py	✔	
Adafruit_*.py	✔	

Table 9-1: Python Module Usage

9.6 Libraries

9.6.1 Installed Libraries

The following libraries are not included in standard Python distribution and must be manually installed:

Library Name	Import Name(s)	Module
NumPy	numpy	IV_Swinger.py, IV_Swinger2_sim.py IV_Swinger_PV_model.py
Matplotlib	matplotlib.pyplot	IV_Swinger.py
pySerial	serial, serial.tools.list_ports	IV_Swinger2.py
Pillow	PIL (Image, ImageTk)	IV_Swinger2_gui.py
SciPy	scipy.optimize	IV_Swinger_PV_model.py
Send2Trash	send2trash	IV_Swinger2_gui.py
pywin32	win32com.client	IV_Swinger2_gui.py

Table 9-2: External Libraries

9.6.2 Standard Libraries

The following libraries are used, but they are included in the standard Python distribution so they do not have to be manually installed. The ones in gray are not used for IV Swinger 2 functionality, but are imported by the IV_Swinger.py module. Some of the import names changed from Python 2 to Python 3; the table lists the Python 3 names.

Import Name	IV_Swinger.py	IV_Swinger_plotter.py	IV_Swinger2.py	IV_Swinger2_gui.py	IV_Swinger2_sim.py	IV_Swinger_PV_model.py	IV_Swinger2_PV_model.py
argparse		✔	✔				
configparser			✔				
csv						✔	
datetime	✔		✔	✔			
difflib			✔				
glob	✔		✔				
inspect			✔				
io			✔				
math	✔		✔				
os	✔	✔	✔	✔	✔	✔	✔
re	✔		✔	✔	✔		
resource				✔			
shutil	✔		✔	✔			

subprocess	✔		✔				
sys	✔		✔	✔			
time	✔		✔				
tkinterr				✔			
tkinter.ttk				✔	✔		
tkinter.filedialog				✔			
tkinter.font				✔			
tkinter.messagebox				✔	✔		
tkinter.scrolledtext				✔	✔		
tkinter.simpledialog[14]				✔			
tkinter.constants				✔			
traceback	✔			✔			
warnings						✔	
threading	✔						
queue	✔						

Figure 9-1: Standard Libraries

9.7 Classes

The IV Swinger 2 Python application code is objected-oriented. Figure 9-2 below is a class diagram showing the relationships between most of the classes in the application. Notably, this diagram does not include the classes in the IV_Swinger2_sim.py, IV_Swinger_PV_model.py or IV_Swinger2_PV_model.py modules which were added later and are not part of the core functionality.

[14] The tkinter.simpledialog module is actually slightly modified in myTkSimpleDialog.py

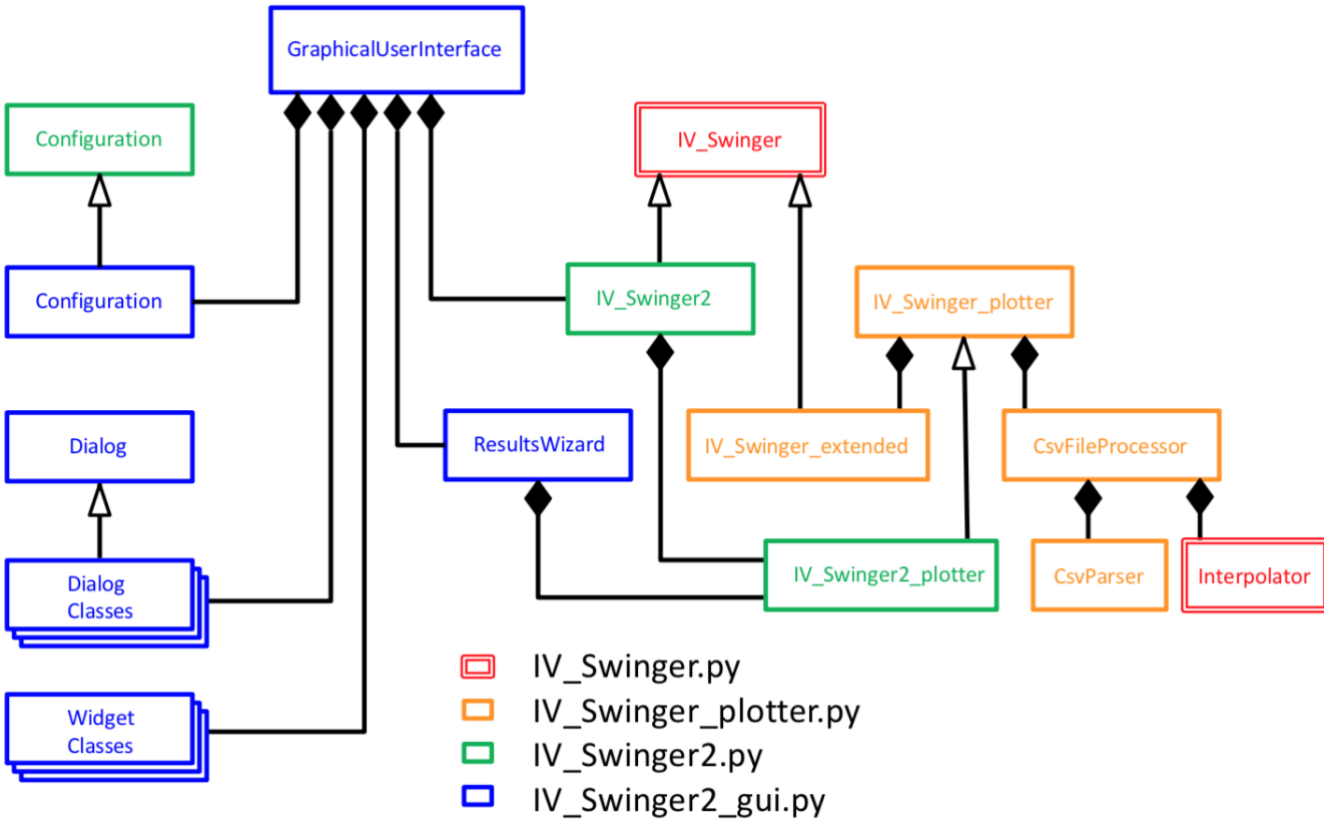


Figure 9-2: IV Swinger 2 Class Diagram

The relationships are indicated using the standard UML notation. Namely, the lines with a hollow triangle on one end indicated an inheritance ("is a") relationship, and the lines with a filled diamond indicate composition ("has a") relationship. So, for example, the IV_Swinger2 class is derived from the IV_Swinger base class. And the GraphicalUserInterface class contains a ResultsWizard object. To keep the chart uncluttered, some classes are omitted, and the dialog and widget classes are not itemized.

The module dependencies can be inferred from Figure 9-2. The IV_Swinger.py module is not dependent on any of the other modules. Standalone, it is only useful on the original IV Swinger (IVS1), however. The IV_Swinger_plotter.py module is dependent on IV_Swinger.py only. It was originally written (and can still be used) as a command-line script to post-process IVS1 results to generate graphs with user-specified preferences (line width, font size, etc.) and to overlay multiple IV curves on the same graph. The IV_Swinger2.py module is dependent on both IV_Swinger.py and IV_Swinger_plotter.py. The IV_Swinger2_gui.py module is dependent on all of the others.

Note that since IV_Swinger2.py is not dependent on IV_Swinger2_gui.py, it can be useful in a non-GUI application. In fact, it has a *main()* function that swings one IV curve, stores the result in the standard (OS-dependent) place, and opens the PDF.

9.7.1 Properties

Access to many of the classes' attributes is provided via Python properties. Property definitions look like method definitions, but are "decorated" with *@property* and/or *@<name>.setter*. This provides "managed access" to the attributes. In the simplest case they are just "getters" and "setters", but they can have more complex actions associated with them than just getting and setting attributes. From the user's point of view, they just look like a variable, i.e. they are accessed without the () that would be used for a

method call. When a property is used on the left side of an equation, the “setter” functionality is invoked. When a property is used on the right side of an equation, the “getter” functionality is invoked. A typical value added of property “setters” is to check the provided value for proper type and other constraints. Some property “getters” perform significant computation based on the values of multiple attributes (or other properties).

9.8 Date/Time Strings

Since IVS1, the Python code has used a yymmdd_hh_mm_ss format for directory names, file names, etc. to indicate the year, month, day, hour, minute and second of an application session, IV curve trace initiation, etc. This is referred to as a date/time string, sometimes abbreviated “dts” in variable names. It is useful because it is reasonably readable and sorts by age (at least until the year 2100).

The IV_Swinger.py module has a class named DateTimeStr that is not shown in the class diagram of Figure 9-2. It is a stateless class (no attributes) that contains nothing other than four static methods: *get_date_time_str()*, *extract_date_time_str()*, *is_date_time_str()* and *xlate_date_time_str()*. This keeps this code in one place, where it can be used by all of the modules.

9.9 Platform-specific Code

The vast majority of the Python code is not dependent on what platform the application is running on. There are a few exceptions, however.

9.9.1 Determining the Platform

The non-GUI code uses the value of sys.platform to determine the platform:

- sys.platform == “darwin”: => Mac
- sys.platform == “win32”: => Windows
- sys.platform == (Other): => Linux (assumed)

The GUI code cares more about the windowing system, so it uses:

- root.tk.call("tk", "windowingsystem") == “aqua”: => Mac
- root.tk.call("tk", "windowingsystem") == “win32”: => Windows
- root.tk.call("tk", "windowingsystem") == “x11”: => Linux

9.9.2 Application Data Directory

The application data directory is where the results, log files, and configuration files are written. The *app_data_dir* property of the IV_Swinger2 class sets it to a platform-dependent value if it is not set by the object creator. Regardless of the platform, the leaf directory is named “IV_Swinger2”. On Windows, the base directory is the value of the APPDATA environment variable. On Mac, some functions imported from the Appkit library previously were used to determine the path to the base directory, but it is now hardcoded to /Users/<user>/Library/Application Support.

The resulting application data directory paths are the following:

- Mac: /Users/<user>/Library/Application Support/IV_Swinger2
- Windows: C:\Users\<user>\AppData\Roaming\IV_Swinger2

The application data directory contains the following directories and files:

- One "yymmdd_hh_mm_ss" directory per run, containing the CSV files with the result data, the configuration file at the time of the run, and the plot image files. This is referred to as the run directory in this document and in the code.
- One "logs" directory, containing the log files for each session
- One "IV_Swinger2.cfg" file containing the current configuration
- One "IV_Swinger2_starting.cfg" file containing the previous configuration
- Zero or one "overlays" directory, containing overlay graphs
- Zero or one "Battery" directory, containing bias battery calibration curves
- Zero or one "pv_spec.csv" file, containing PV modeling specifications
- Zero or one "pv_spec_bak.csv" file, containing the previous pv_spec.csv

9.9.3 System File Viewer

All of the supported platforms "know" how to open certain types of files based on the file's type or extension. When the user double-clicks on a file in the File Explorer (Windows) or Finder (Mac), the OS opens the file using the associated application. For example, a file with a .txt extension will be opened with a text editor. A PDF will be opened with a program that knows how to display PDFs (Acrobat Reader, Preview). The user may have some say about what application is used for a given file type.

The IV Swinger 2 application needs to be able to open text files and PDFs for the user to look at. It leaves the choice of the viewer application to the OS. Unfortunately, this has to be done slightly differently depending on the platform. The global function *sys_view_file()* in the IV_Swinger2.py module uses the code from a StackOverflow answer to do this.

9.9.4 Platform-specific GUI Code

For the most part, the platform-specific GUI differences do not map to code differences since the tkinter/tkinter.ttk framework takes care of them. There are a few notable exceptions:

- Position of OK and Cancel buttons in dialog windows: Mac has Cancel to the left of OK, both in the lower right corner; Windows has OK in the lower left corner and Cancel in the lower right corner.
- Mac has an "IV Swinger 2" menu, whereas Windows has an "About" menu
- Mac has a (pretty useless) "Window" menu, whereas Windows has no equivalent
- Both have "Help" menus, but the code to create them is different

9.10 Logging

Although Python includes a logging facility, the IV Swinger code does not currently use it (perhaps it should).

Not shown in the class diagram of Figure 9-2 is the PrintAndLog class, defined in the IV_Swinger.py module. This is a very simple class with only one class variable: *log_file_name*. It has two methods: *log()* and *print_and_log()*. The former writes a message to the log file. The latter first prints the message to the screen (if there is one) and then calls *log()*.

The IV_Swinger2 class has a *configure_logging()* method that is called at initiation time. It uses the current date/time string to generate the log file name and sets the PrintAndLog *log_file_name* class variable to that name. It then creates an instance of PrintAndLog and assigns it to its *logger* attribute. Since the *logger* attribute is shared with its base class, IV_Swinger, the same logger is used by both classes and all log messages go to the same file.

The IV_Swinger2_gui class uses its IV_Swinger2 object's logger when it needs to write to the log file.

9.11 Configuration

Configuration is a feature added for IVS2; it was not supported for IVS1. It provides the mechanism for settings/options/preferences/knobs to be remembered from one invocation of the application to the next.

The Configuration class provides support for saving configuration values to a file and restoring them from that file using Python's configparser module. The saved/restored values map directly to IV_Swinger2 and IV_Swinger2_gui class properties.

9.11.1 Base Class

The Configuration base class is defined in IV_Swinger2.py. It deals only with configuration values that map to properties in the IV_Swinger2 class.

An object of the Configuration base class operates on an object of the IV_Swinger2 class. It is passed the IV_Swinger2 object name when it is instantiated.

The following configuration file sections are defined:

- [General]
- [USB]
- [Calibration]
- [Plotting]
- [Arduino]
- [PV Model]

Each section contains one or more "name = value" entries.

9.11.2 Derived Class (GUI)

The Configuration class in IV_Swinger2_gui.py is derived from the base class, and extends it to add the following configuration file section:

- [Looping]

This section is not included in the base class because the looping feature is implemented in the GUI.

An object of the Configuration derived class operates on an object of the IV_Swinger2_gui class and on the IV_Swinger2 object it instantiates. It is passed the IV_Swinger2_gui object name when it is instantiated.

9.11.3 Basic Configuration Functionality

The most basic actions performed by the Configuration class are reading the values from the configuration file and writing values to the file. The *get()* method does the former and the *save()* method does the latter. This is shown below in Figure 9-3.

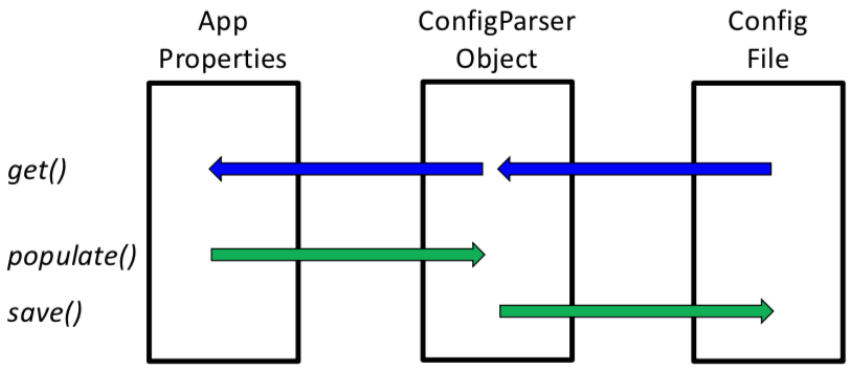


Figure 9-3: Configuration Data Movement

The ConfigParser object is an intermediary between the file and properties.

If the file does not exist when the *get()* method is called, the method creates it by first calling the *populate()* method to fill the ConfigParser object with the values from the properties, and then calling the *save()* method to create the file with those values. If the file does exist, the *get()* method uses the *read()* ConfigParser method to read and parse the configuration data from the file into the ConfigParser object, and then it applies those values to the associated properties (using the *apply_*()* methods).

9.11.4 Additional Configuration Functionality

9.11.4.1 Snapshots

In addition to the primary ConfigParser object instantiated by the Configuration class (instance name: *cfg*), there is a second one that is named *cfg_snapshot*. This "snapshot config" uses the same config file. There are *get_snapshot()* and *save_snapshot()* methods that are similar to the *get()* and *save()* methods, except that the *get_snapshot()* method does not update the app properties. The purpose of this is to be able to capture the configuration before taking actions that modify it, and then be able to revert to the previous configuration later.

9.11.4.2 Support for Reprocessing Old Results

It is useful to be able to reprocess results from IV curves that were traced in the past. By default, the regenerated graph should look identical to the one that was generated when the curve was traced. This

necessitates saving not only the curve data, but also the configuration at the time of the run. The *copy_file()* method does this by copying the current config file to a specified directory (i.e. the one where the run's results are saved). The *get_old_result()* method is similar to the *get()* method, but it reads from a specified "old" config file and updates only the properties that are relevant to reprocessing old results (General, Calibration, Plotting). The *merge_old_with_current_plotting()* method calls the *get_old_result()* method, but discards the Plotting section values and replaces them with their values from the current config. This is useful for a batch update to a common set of plotting preferences, while preserving everything else.

9.11.4.3 Debug Features

A method named *save_starting_cfg_file()* simply copies the current config file to a "starting config" file. It is called when the Configuration object is created. The method named *log_cfg_diffs()* compares that file with the original and reports what did and did not change to the log file. This can be very useful for debugging, to see what changed in the configuration during an invocation of the app.

Another method, *cfg_dump()* is also used for debugging. It prints all of the config values to the log file at the time it is called.

9.12 Arduino Interaction

The ability to interact with the Arduino is prerequisite to being able to swing an IV curve. This section describes the application code that deals with the interaction with the Arduino sketch, other than the swinging of the IV curve which is the topic of the next section. All such code is contained in the IV_Swinger2 class. The IV_Swinger class knows nothing about Arduino, and the IV_Swinger2_gui class interacts with the Arduino only via its instantiated IV_Swinger2 object.

9.12.1 PySerial

The PySerial library provides access to the laptop's USB ports, taking care of platform-specific (Windows/MacOS/Linux) details. The two modules that are imported from this library are "serial" and "serial.tools.list_ports".

9.12.2 Finding the Arduino

The first step is identifying which of the laptop's USB ports is connected to the IVS2 Arduino. The IV_Swinger2 class's *find_serial_ports()* method uses the *comports()* method from serial.tools.list_ports to list all of the serial ports. The *find_arduino_port()* method searches the list of ports for one that has the string "uino" in its metadata. It may or may not find one, either because no Arduino is connected to the laptop, or because it has a more generic name without that string in it. The *usb_port* property is set to the name of the USB port connected to the Arduino. This property can be set to a value without calling the *find_arduino_port()* method. For example, the config file saves the port that was last used, and that port is assumed to be the one that will be connected to the IVS2 Arduino the next time the application is run.

The *usb_port_disconnected()* method uses the *comports()* method from serial.tools.list_ports to list all of the serial ports, and then it searches them to see if one matches the *usb_port* property. If the cable has

become disconnected, it will not be found and the *usb_port_disconnected()* method returns a value of True. If it is found, a value of False is returned.

9.12.3 Resetting the Arduino and Establishing Communication

The *reset_arduino()* method causes the Arduino to be reset, just as if the physical button had been pushed. This starts the Arduino sketch from the beginning, running the *setup()* function.

The reset is accomplished by opening the USB port that the Arduino is connected to. If the port was already open, it closes it before opening it. Opening the port is accomplished by creating an object of PySerial's "Serial" class. The Arduino reset is a side-effect of opening the USB port.

After the USB port is successfully opened, a bi-directional buffered text stream object is created around the raw USB byte stream, using io.TextIOWrapper, as recommended in the PySerial documentation.

9.12.4 Sending Messages

The *send_msg_to_arduino()* method uses the *write()* method of the io.TextIOWrapper object to send a string to the Arduino sketch. Since the Arduino sketch has a fixed-size array to store incoming messages, the *send_msg_to_arduino()* method checks that the string provided by the caller does not exceed that length.

9.12.5 Receiving Messages

The *receive_msg_from_arduino()* method uses the *readline()* method of the io.TextIOWrapper object to receive a string from the Arduino sketch. This method is only called when a message is expected. The message may not arrive immediately, however. If no message is present, *readline()* returns a zero length string. In that case, *readline()* is called repeatedly until a non-zero length string is returned. After 50 tries, however, a timeout is detected.

9.12.6 Arduino Handshake

The *wait_for_arduino_ready_and_ack()* method is called after the Arduino is reset. It performs the initial handshake with the Arduino sketch as described in Section 8.2.1on page 95 and shown in Figure 8-1 on page 97.

The *receive_msg_from_arduino()* method is called to capture the first message from the Arduino. Modern versions of the Arduino sketch start with the string that contains the version number of the sketch and then send the "Ready" message. Ancient versions of the sketch just start with the "Ready" message. For backward compatibility, the *wait_for_arduino_ready_and_ack()* method works with either. If the version number string is received, it parses that with the *get_arduino_sketch_ver()* method and then calls the *receive_msg_from_arduino()* method again, to wait for the "Ready" message.

When the "Ready" message is received from the Arduino, the *arduino_ready* property is set to True so other code knows that the sketch is ready to swing IV curves.

Next, the *wait_for_arduino_ready_and_ack()* method calls the *send_config_msgs_to_arduino()* method, described in Section 9.12.8 below. Then it calls the *request_eeprom_dump()* method, described in Section 9.12.9.

The initial handshake concludes by the *wait_for_arduino_ready_and_ack()* method sending a "Ready" message to the Arduino, using the *send_msg_to_arduino()* method.

9.12.7 Arduino Sketch Compatibility

The initial handshake is compatible with all versions of the Arduino sketch, and must remain so. However, since the version of the sketch is included in the first string received from it, code running after that can use the version to determine whether certain features are supported or not by the sketch that it is communicating with.

The *compare_arduino_sketch_ver()* method and wrapper methods *arduino_sketch_ver_[lt|eq|gt|le|ge]()* provide all of the possible comparisons that could be needed to determine compatibility. There are further wrappers (implemented as properties) to test for specific features. For example:

```
# --------------------------------
@property
def arduino_sketch_supports_ssr_adv_current_cal(self):
    """True for Arduino sketch versions that have code to support
       SSR advanced  current calibration.
    """
    return self.arduino_sketch_ver_ge("1.3.8")
```

This property has a value of "True" if the Arduino sketch is version 1.3.8 or higher (ge=greater-than-or-equal), and a value of "False" otherwise.

9.12.8 Sending Configuration Messages

The *send_one_config_msg_to_arduino()* method sends one "Config" message to the Arduino sketch using the *send_msg_to_arduino()* method. It then calls the *receive_msg_from_arduino()* method until it receives a "Config processed" message, indicating that the sketch processed the config message.

The *send_config_msgs_to_arduino()* method sends all of the defined config messages that map to IV_Swinger2 class property values with the value of their respective property. However, as an optimization, it skips ones that it has already sent if the property value has not changed since the last time it sent that config value. The method has a *write_eeprom* parameter, that the caller sets when it requires the WRITE_EEPROM config to be sent for each of the valid EEPROM locations (in addition to sending the other config values).

9.12.9 Reading and Writing EEPROM

The Arduino EEPROM is used primarily to store calibration values so that they "follow" the IV Swinger 2 hardware even if that hardware used with different laptops.

Reading the EEPROM is done by sending the DUMP_EEPROM config message to the Arduino. This is done by the *request_eeprom_dump()* method. There is no support (and no need) for reading a single EEPROM location. The *request_eeprom_dump()* method calls the *receive_msg_from_arduino()* method until it receives a "Config processed" message, which comes after all of the valid address/value pairs are received. For each EEPROM address/value message that is received from the Arduino sketch, *request_eeprom_dump()* calls *process_eeprom_value()* which looks at the address and sets the associated IV_Swinger2 property to the value that was read from that address.

Writing the EEPROM is done by the *send_config_msgs_to_arduino()* method when it is passed *write_eeprom*=True, as described in the previous section.

The *invalidate_arduino_eeprom()* method writes a value of 0 to EEPROM address 0. This overwrites the "magic number" at that location, which effectively invalidates all of the other locations. The *restore_arduino_eeprom()* method writes the magic number to EEPROM address 0.

The *write_relay_active_high_val_to_eeprom()* method sends one WRITE_EEPROM config message to write the value of the *relay_active_high* property to EEPROM location 44.

9.13 Swinging an IV Curve

The IV_Swinger2 class has a method named *swing_curve()*. Of course, the actual swinging of the curve is performed by the Arduino sketch. The IV_Swinger2 *swing_curve()* method does the following:

- Sets up for run
- Triggers the Arduino
- Receives the data points from the Arduino
- Writes the ADC values to a CSV file
- Processes ADC values
- Plots the results

This section describes all but the plotting, which is covered in the next section.

9.13.1 Pre-Swing Setup Tasks

Before triggering the Arduino to swing the curve, the following prerequisite tasks are performed:

- Get the current date/time string, retrying until the second rolls over if it is the same as the previous run. This limits the rate to one curve per second, which is a hardware requirement.
- Call *create_hdd_output_dir()* to create the run directory, named for the date/time string
- Write "Swing!" message with run directory name to the log file
- Call *get_csv_filenames()* method to set properties to the names of the CSV files, using the date/time string
- If necessary, reset the Arduino and perform the initial handshake
- Send config messages to Arduino for any values that have changed

9.13.2 Triggering the Arduino

Triggering the Arduino to swing the IV curve is done by sending it a "Go" message. The Arduino sketch swings the curve, and then sends the results.

9.13.3 Receiving the Data from Arduino

The *receive_data_from_arduino()* method is a wrapper around the *receive_msg_from_arduino()* method. It loops receiving Arduino messages until it sees an "Output complete" message. It then loops through the list of messages that were received, parsing each one.

Data point messages looks like the following:

```
Isc CH0:0 CH1:787
0 CH0:99 CH1:779
1 CH0:105 CH1:775
...
64 CH0:465 CH1:15
65 CH0:465 CH1:5
Voc CH0:819 CH1:0
```

The CH0 value is the ADC reading on the voltmeter channel, and the CH1 value is the ADC reading on the ammeter channel. A tuple containing these two values is appended to the *adc_pairs* list, which is a property of the IV_Swinger2 class.

In addition to the data point messages, the *receive_data_from_arduino()* method parses and processes other messages that the Arduino sketch may or may not send before the "Output complete" message, namely:

- A "Polling for stable Isc timed out" message
- Temperature sensor and pyranometer messages
- A bandgap ADC message

The bandgap ADC message is used to update the *adc_vref* property using the *set_vref_from_bandgap()* method. This is part of the feature added in software release v2.6.0 where the ATmega328's internal 1.1V bandgap voltage is used to measure the reference voltage. The Arduino sketch reads the bandgap ADC value for every curve, so this means *adc_vref* is updated for each.

9.13.4 Writing the ADC Pairs to a CSV File

The *write_adc_pairs_to_csv_file()* method writes each pair of ADC readings to a CSV file in the output directory. This file is not used for the current run, but may be used later for debug or for regenerating the IV curve with different plotting options. It is the only history of the raw readings other than the captured Arduino messages in the log file.

9.13.5 Processing the ADC Values

The *process_adc_values()* method does the following:

- Determines ADC Offset Values
- Performs sanity checks
- Applies battery bias, if enabled
- Applies calibration adjustments
- Performs other corrections
- Converts ADC values to volts/amps/watts/ohms
- Writes converted values to CSV file

9.13.5.1 ADC Offset Values

The *get_adc_offsets()* method analyzes the ADC pairs to determine (or infer) the "noise floor" for each ADC channel, i.e. the minimum ADC value. This is called the ADC "offset", which is a misleading name that preceded the understanding that it is really a noise floor.

9.13.5.2 Sanity Checks

The *adc_sanity_check()* method checks the ADC pairs to make sure V_{OC} is not zero volts and I_{SC} is not zero amps.

9.13.5.3 Battery Bias

This only applies to the cell version, when a bias battery is being used. The *apply_battery_bias()* method is called to subtract the battery bias from the measured points.

9.13.5.4 Calibration Adjustments

The *calibrate_adc_pairs()* method applies the voltage and current calibrations to the ADC values.

9.13.5.5 Other Corrections

The *correct_adc_values()* method removes errors from the ADC values. This consists of the following corrections:

- Combine points with same voltage (use average current)
- Zero out the CH1 (current) value for the V_{OC} point
- Remove the I_{SC} point if it is not reliable
- Replace the I_{SC} point with a better extrapolation than the Arduino code was capable of
- Apply a noise reduction algorithm
- Adjust ADC values voltages to compensate for V_{OC} shift

Each correction is independently enabled/disabled by a method input parameter. Some deserve some more explanation, and that is given in the following subsections.

9.13.5.5.1 I_{SC} Extrapolation Algorithm

The Arduino script doesn't really know what the I_{SC} current is because the hardware cannot create a true short circuit. The I_{SC} value that it reports is the current of the measured point with the lowest voltage, after the current value has stabilized. This is often not a very good extrapolation.

The *create_new_isc_point()* method replaces the I_{SC} point with a new "better" one. The algorithm starts by analyzing the ADC values looking for where the curve begins to deflect downward. It then uses the points before that to determine where the curve would intersect with the vertical axis.

9.13.5.5.2 Noise Reduction Algorithm

Due to noise in the hardware circuitry, the curve formed by the measured (I,V) values is usually somewhat “bumpy”. This is mostly an aesthetic issue, but it can slightly affect the calculation of the maximum power point (MPP).

The *noise_reduction()* method smooths out the "bumps" in the curve. The trick is to disambiguate between deviations (bad) and inflections (normal). For each point on the curve, the rotation angle at that point is calculated. If this angle exceeds a threshold, it is either a deviation or an inflection. It is an inflection if the rotation angle relative to several points away is actually larger than the rotation angle relative to the neighbor points. Inflections are left alone. Deviations are corrected by replacing them with a point interpolated (linearly) between its neighbors. This algorithm may be performed incrementally, starting with a large threshold and then dividing that threshold by some amount each time - this provides better results because the larger deviations will be smoothed out first, so it is clearer what is a deviation and what is and what isn’t.

This algorithm works very nicely for this application. I came up with it myself, but I don’t know if someone else may have beaten me to it. It could be improved slightly by not making the correction a linear interpolation, but instead by choosing a point whose rotation angle relative to its immediate neighbor is equal to the rotation angle relative to the points several points away.

9.13.5.5.3 V_{OC} Shift (Overshoot) Compensation Algorithm

Section 4.2.1.5 on page 47 describes how the current drawn by the electromagnet in the EMR causes a droop in the +5V reference voltage, having the effect that the voltage measurements are too high when the relay is active. This manifests itself as an “overshoot”, where the tail of the IV curve is at a higher voltage than the measured V_{OC} point. The voltage of the V_{OC} point is correct, because it is measured when the relay is inactive.

The reason for the overshoot was not initially understood, but the fact that the voltage of the V_{OC} point was correct was known because it could be verified (and calibrated) with an external DMM. Clearly the tail of the curve should line up with the V_{OC} point. The obvious solution was to simply scale the voltages of all points (except the V_{OC} point) so that this would be the case. Fortunately, once the problem was understood, this algorithm turned out to be valid.

The SSR-based design does not suffer from this overshoot because SSRs do not draw current when they are activated. For some not-yet-understood reason, there sometimes seems to be the opposite effect (slight undershoot).

Regardless of the reason, the IV curve is clearly more accurate when the tail lines up perfectly with the measured and trusted V_{OC} point. Undershoot is just another name for negative overshoot, so the same algorithm works for that too.

The *v_adj()* method returns a single scaling value. This is just the ratio between the voltage at the point where the tail of the curve intersects the V-axis and the voltage of the V_{OC} point. The tricky part is determining where the intersection point is. Simply using the last two points of the tail isn’t always good because they can be very close together, and for a number of reasons may not determine a line that is pointing in the same direction as the rest of the tail. Instead, the intersection points of the lines determined by the last point and each the four preceding points are all calculated, and the average

intercept is used. Some of these intercepts may be excluded from the average, however, if they differ by too much from the others.

9.13.5.6 Conversion to Volts/Amps/Watts/Ohms

The *convert_adc_values()* method converts the calibrated and corrected ADC values to volts, amps, ohms, and watts.

It uses two properties, *i_mult* and *v_mult*, which when multiplied by the respective ADC values, result in the current and voltage of a point.

9.13.5.6.1 i_mult and v_mult properties

Both *i_mult* and *v_mult* are derived properties that perform a calculation before returning their value.

Both use the *adc_inc* property, which is also derived and is equal to the volts per ADC increment. That is equal to the reference voltage (nominally 5.0V, but subject to calibration) divided by the range of the ADC (4096).

The *i_mult* property returns (*adc_inc* / *amm_op_amp_gain* / *amm_shunt_resistance*). The *amm_op_amp_gain* and *amm_shunt_resistance* properties are inherited from the IV_Swinger class and are the value of the ammeter op amp gain (based on resistor values Rf and Rg) and the shunt resistor resistance.

The *v_mult* property returns (*adc_inc* / *vdiv_ratio*). The *vdiv_ratio* property is the voltage divider ratio, based on resistor values R1 and R2.

9.13.5.6.2 Series Resistance Compensation

If the *series_res_comp* property is non-zero, the calculated voltage is increased by the amps times the value of *series_res_comp* (which can be negative).

9.13.5.6.3 Data Points

After calculating the current and voltage, the *convert_adc_values()* method also calculates watts (volts * amps) and ohms (volts / amps). It appends a tuple with the four values to the IV_Swinger2 class's *data_points* list property. It also writes a line to the log file with the four values.

The format of the *data_points* property, which is a list of (amps, volts, ohms, watts) tuples was chosen for compatibility with methods in the IV_Swinger base class, allowing their re-use.

9.13.5.7 Writing Converted Values to Data Points CSV File

The *write_csv_data_points_to_file()* method is inherited from the IV_Swinger class (defined in IV_Swinger.py). It opens the specified CSV file for writing, and then writes one line with the headings followed by the volts, amps, watts, and ohms of each point in the *data_points* list.

Because this CSV file is formatted exactly the same for IVS1 and IVS2, the same tools can be used to plot the IV curves for both generations.

9.14 Plotting

The code for plotting the IV curves is heavily shared/reused from the original IV Swinger code. All the code that is needed to plot a single IV curve is contained in the IV_Swinger.py module. The IV_Swinger_plotter.py module was originally written as a standalone command-line utility to enable the IVS1 user to generate new IV curves from existing data points CSV files, with some added features such as overlays. But it can also be imported by another module so its classes and capabilities can be used, and that is exactly what the IV_Swinger2.py module does.

Figure 9-4 below is a subset of the class diagram in Figure 9-2 on page 121. It shows the relationships between the classes that are involved in IV Swinger 2 plotting.

This section will start by describing the plotting support in the IV_Swinger.py module. Then it will cover the plotting extensions that are added by the IV_Swinger_plotter.py module. Finally, it will describe how the IV_Swinger2.py module uses the plotting code in the two older modules. It will not discuss the GUI, which sits on top of all of this.

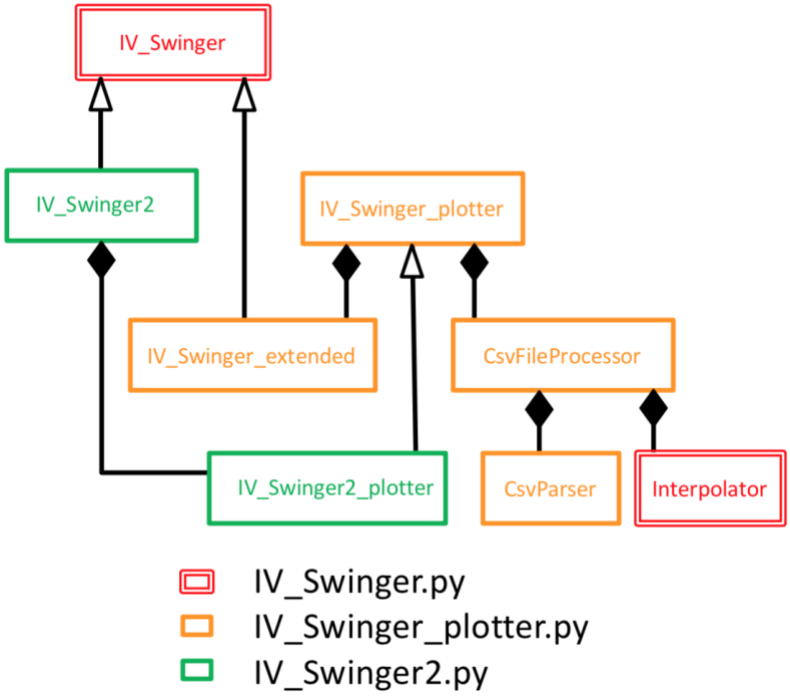


Figure 9-4: Plotting Class Diagram

9.14.1 IV_Swinger.py Module Plotting

The original IV Swinger used an external plotting utility called gnuplot to generate PDF graphs of the IV curves. When the IV_Swinger_plotter.py utility was added, support for an alternate Python-based plotting utility called pyplot (part of the matplotlib library) was added to the IV_Swinger.py module. The gnuplot option is still in the code, but it is not used by the IV Swinger 2 code.

The ultimate output of pyplot is a PDF, GIF or PNG image file. Before that image file can be generated, the following elements have to be added to the figure:

- The measured IV curve data points
- The interpolated curve connecting the measured points
- (Optionally) the power curve
- The I_{SC}, MPP, and V_{OC} points and labels
- Title, legend, axis labels, axis ticks and grid lines

In addition to adding the elements listed above, there are several characteristics of the figure as a whole and of those elements that must be defined, including:

- Figure size, axis ranges, fonts, size/style of plotted points, width/style of plotted lines, colors, resolution (DPI)

Pyplot is the state-based interface to matplotlib. "State-based" means that a figure is created by calling pyplot functions one at a time to incrementally change that figure. This is similar to how MATLAB works. Matplotlib does have an object-oriented API that might have been better to use than the pyplot API, but the pyplot API was more closely matched to gnuplot.

Regardless of whether it is for runtime IVS1 plotting, command-line plotting using the IV_Swinger_plotter.py utility, runtime IVS2 plotting or post-processed IVS2 plotting, the following sequence is performed using the IV_Swinger.py module code described in this section:

1. Generate interpolated data points
2. Generate a plotter data points file
3. Generate an image file

9.14.1.1 Interpolation

IVS1 captured far fewer measured points than IVS2 does. It was important to interpolate a "smooth" curve between the measured points for two reasons:

- Accurate estimation of the maximum power point (MPP)
- Aesthetics

A linear interpolation would look ugly and the MPP would not be accurate because the knee of the curve would likely be "sliced off", resulting in a lower than actual MPP power. Instead, a "spline" interpolation was required, and Catmull-Rom was chosen[15].

With the far greater number of measured points generated by IVS2, there is little difference between linear interpolation and spline interpolation aesthetically or for MPP interpolation. Linear interpolation takes less time and is the default. The spline interpolation is still supported, however.

[15] There is much more detail about this in "IV Swinger: Design, Construction and Operation" Section 8.4.6.3.9, p. 92.

The Interpolator class in the IV_Swinger.py module performs the interpolation and identification of the MPP. When an Interpolator object is created, it is passed the list of measured data points. No interpolation is performed until one of the Interpolator class properties is accessed. Note that these are properties (getter only), not methods, but they perform a lot of computation. The properties are:

- *linear_interpolated_curve*
- *spline_interpolated_curve*
- *linear_interpolated_mpp*
- *spline_interpolated_mpp*

The first two return a list of points that includes the measured points, but also includes points interpolated between those points. In the case of the spline interpolation, up to 100 points are added between each measured point, depending on how close together the measured points are. In the case of linear interpolation, points are added only for the segments before and after the measured point with the highest power. This is because the only value added by linear interpolation is to more accurately locate the MPP, which is in one of those segments.

The other two properties use the first two properties to search for the interpolated point with the highest power, and they return the tuple for that point.

9.14.1.2 Generating a Plotter Data Points File

Not to be confused with the data points CSV file, the "plotter data points" file is a temporary file that contains both the measured points and the interpolated points. It is not a CSV file. It would not be necessary if pyplot were the only plotter supported, but it is needed for gnuplot, which was the original IV Swinger plotter and is still supported for IVS1.

The format of the plotter data points file is based on gnuplot requirements. Each data point line in the file has three values, separated by a space character. The first is volts, the second is amps, and the third is watts. Two blank lines indicate the start of a new "data set". This is used to separate the interpolated points from the measured points. Figure 9-5 below shows an example of a plotter data points file. All of the middle lines of each data set are hidden with "….".

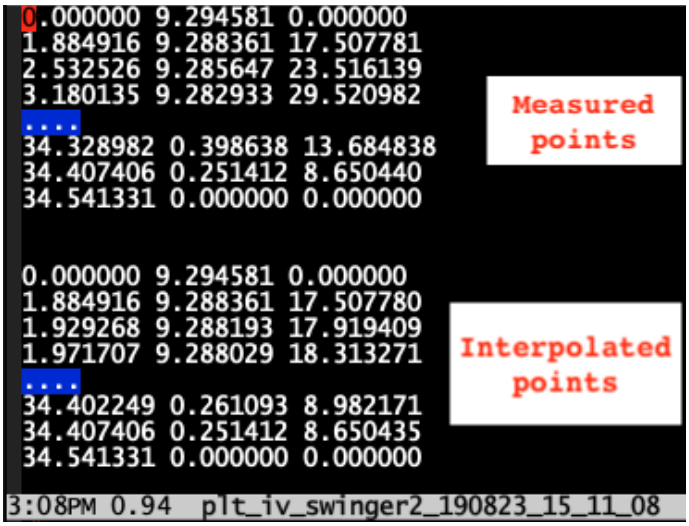


Figure 9-5: Plotter Data Points File

The IV_Swinger class's *write_plt_data_points_to_file()* method is used to create a plotter data points file. It is passed a file name, a *data_points* list of (amps, volts, ohms, watts) tuples, and an indication whether the data points are a new data set. This method is intended to be called twice – first with an empty or non-existent file and the list of measured data points, and the second with the list of interpolated data points, which will be appended to the existing file created by the first call.

9.14.1.3 Generating a Plotter Image File

The IV_Swinger class's *plot_with_plotter()* method takes as input:

- A list of plotter data points files
- A list of the I_{SC} amps values
- A list of the V_{OC} volts values
- A list of the MPP amps values
- A list of the MPP volts values
- The name of the output image file

When a single curve is being plotted, each of the lists has a single entry only. When an overlay is being plotted, the lists can have up to 8 entries (and all must be the same length).

The *plot_with_plotter()* method calls either the *plot_with_gnuplot()* or the *plot_with_pyplot()* method to create the image file with the plot. Since IV Swinger 2 does not use gnuplot, only the latter is relevant to this document.

The *plot_with_pyplot()* method calls the following methods:

- *set_figure_size()*
- *set_figure_title()*
- *set_x_label(), set_y_label()*
- *set_x_range(), set_y_range()*
- *set_x_ticks(), set_y_ticks()*
- *display_grid()*
- *plot_points_and_curves()*
- *plot_labeled_points()*
- *shade_v_sat_area(), shade_i_sat_area()*
- *display_legend()*
- *adjust_margins()*
- *print_img_to_file()*

The *plot_points_and_curves()* method calls several sub methods to read the plotter data points file and plot the measured points and interpolated curve. It also optionally determines and plots the power curve.

The *plot_labeled_points()* method calls several sub methods to plot and label the I_{SC}, MPP and V_{OC} points.

The other methods are simple, and their names indicate their purpose.

9.14.2 IV_Swinger_plotter.py Module Plotting

As mentioned earlier, the IV_Swinger_plotter.py module was written as a command-line utility to generate plots from the CSV files containing the measured data points from a previous IV Swinger run. It added the following plotting enhancements:

- The ability to add the power curve to the same graph
- The ability to overlay up to 8 IV curves on a single graph
- Multiple options to customize the appearance of the graph:
  - Chart title
  - Curve names in legend
  - Size of the plot, font sizes, point sizes, line width
  - Fancy labels on Isc, Voc, and MPP
  - Linear or smooth interpolation

To be clear, most of these enhancements did require added support in the plotting code in the IV_Swinger.py module but none of them was used for the runtime generation of IV curve plots by Python running on the Raspberry Pi. The IV_Swinger_plotter.py utility was intended to be used on a desktop or laptop computer to post-process the results after they were transferred to that computer on the USB thumb drive that was written by the Raspberry Pi.

IV Swinger 2 needs all of these features. While it would have been possible to use IV_Swinger_plotter.py as an external utility, it was cleaner to import it as a module and use its classes and methods directly.

9.14.2.1 Classes

The IV_Swinger_plotter.py module defines the following classes:

- PrintAndOrLog
- CommandLineProcessor
- CsvParser
- IV_Swinger_extended
- CsvFileProcessor
- IV_Swinger_plotter

9.14.2.1.1 PrintAndOrLog Class

The PrintAndOrLog class is used internally by the module to print messages to the console and/or to log them using an IV_Swinger.py PrintAndLog logger object. When used as a standalone utility, there is no logger, so the messages can only be printed.

9.14.2.1.2 CommandLineProcessor Class

The CommandLineProcessor class is used only when IV_Swinger_plotter.py is used as a standalone utility. It uses the Python argparse library to parse the command-line arguments. The results are placed in an argparse.Namespace object that is used by the other IV_Swinger_plotter.py classes. When

IV_Swinger_plotter.py is imported as a module, the importing code fabricates an argparse.Namespace object with the equivalent values to emulate its having been run from the command line.

9.14.2.1.3 CsvParser Class

The CsvParser class is passed the name of one data points CSV file at object creation time. It has a single property called *data_points* that opens the CSV file and parses the voltage, current, power and resistance values from each line and returns a list of tuples. The *data_points* property is the inverse of the *write_csv_data_points_to_file()* method that was used to create the data points CSV file in the first place. Currently, the CsvParser class is used only by the CsvFileProcessor class, which is described next.

9.14.2.1.4 CsvFileProcessor Class

The CsvFileProcessor class is passed a list of one or more data points CSV files at object creation time. It is also passed the argparse.Namespace object containing all of the actual or fabricated command line arguments, and it is passed an IV_Swinger_extended object. The *proc_all_csv_files()* method is called at initialization; it calls the *proc_one_csv_file()* for each CSV file. The *proc_one_csv_file()* method creates a CsvParser object to read the data points from the CSV file and then uses the Interpolator to generate the plotter data point file and compute or extract the I_{SC}, V_{OC}, and MPP values. It populates the results into the following properties:

- *plt_data_point_files*
- *plt_isc_amps*
- *plt_voc_volts*
- *plt_mpp_amps*
- *plt_mpp_volts*

All of these properties are lists, where each entry in the list is the value for one of the CSV files. The *proc_one_csv_file* method appends to these lists each time it is called. If there is only one CSV file, the lists all have lengths of 1.

9.14.2.1.5 IV_Swinger_extended Class

The IV_Swinger_extended class is derived from the IV_Swinger base class. It is extended to override the base class's *extrapolate_isc()* method (not used for IVS2) and to add a *plot_graphs()* method.

The *plot_graphs()* method is passed a CsvFileProcessor object, populated from one or more CSV files. In the overlay case, the *plot_graphs()* method generates a single image file with multiple curves, by passing the *plot_with_plotter()* method multiple plotter data point files, I_{SC} amp values, V_{OC} volts values, and MPP amps/volts values. In the non-overlay case, it generates a separate image file for each data points CSV file it was passed. IVS2 only populates the CsvFileProcessor object from multiple CSV files for the overlay case.

The other plotting support is inherited from the base IV_Swinger class, namely:

- Generating Plotter Data Points File
- Generating Plotter Image File

9.14.2.1.6 IV_Swinger_plotter Class

The IV_Swinger_plotter class uses its other classes to implement the command-line plotting utility. Its *run()* method does the following:

- Creates a CommandLineProcessor object
- Creates an IV_Swinger_extended object and calls the *set_ivs_properties()* method to set its properties
- Calls its *check_names()* method to check for user errors on the command line
- Creates a CsvFileProcessor object to process all CSV files
- Calls the IV_Swinger_extended object's *plot_graphs()* method

9.14.3 IV_Swinger2.py Module Plotting

The IV_Swinger2.py module mostly leverages the plotting support from the IV_Swinger.py and IV_Swinger_plotting.py modules. It extends the IV_Swinger_plotter class, and the *plot_results()* method of the IV_Swinger2 class uses that extended plotter class.

Now may be a good time to look back at the class diagram in Figure 9-4 on page 134. All of the classes in that diagram have been discussed already except the ones in green.

9.14.3.1 IV_Swinger2_plotter Class

The IV_Swinger2.py module extends the IV_Swinger_plotter base class to define the IV_Swinger2_plotter class. The extensions are:

- Adds properties: *csv_files, plot_dir, args, current_img, x_pixels, generate_pdf, curve_names, title, fancy_labels, label_all_iscs, label_all_vocs, label_all_mpps, mpp_watts_only, linear, overlay, plot_power, font_scale, line_scale, point_scale, v_sat, i_sat, logger, ivsp_ivse and csv_dirs*
- Adds methods: *set_default_args(), plot_graphs_to_pdf(), plot_graphs_to_gif() and add_sensor_info_to_curve_names*()
- Overrides *run()* method

Most properties are for the creator of an IV_Swinger2_plotter object to control its behavior. A few are used internally by the class. One (*current_img*) is used by the object creator to retrieve the file name of the created GIF image file.

9.14.3.1.1 Curve Names

The *curve_names* property is an 8-entry list of the names that will be listed in the legend to identify the curve(s). By default, the first entry is "Interpolated IV Curve" and the others are the Python None value. That is what is used for a normal, non-overlay case. Before requesting the plot generation, the object creator fills the *curve_names* property with the appropriate names for the curves in the overlay case.

The *add_sensor_info_to_curve_names*() method modifies curve names to append the irradiance and/or temperature values to the curve names if the directory containing the CSV file also contains a sensor info file, and it contains irradiance and/or temperature info.

9.14.3.1.2 Fabrication of Command Line Arguments

As described in Section 9.14.2.1.2 on page 138, the IV_Swinger_plotter base class uses the argparse library to parse command line options into an argparse.Namespace object that is used by the other IV_Swinger_plotter.py classes. Since the IV_Swinger2_plotter class is not used in a command line context, an argparse.Namespace object must be created and populated programmatically.

9.14.3.1.3 Plotting to PDF

The *plot_graphs_to_pdf()* method is passed an IV_Swinger_extended object and a CsvFileProcessor object. It calls the base class's *set_ivs_properties()* method to configure the IV_Swinger_extended object, using the fabricated command line arguments. Then it calls the IV_Swinger_extended object's *plot_graphs()* method to create the PDF file (which is the default format).

9.14.3.1.4 Plotting to GIF

The *plot_graphs_to_gif()* method is similar to the *plot_graphs_to_pdf()* method. However, when it calls the IV_Swinger_extended object's *plot_graphs()* method, it sets the "png" option so a PNG file is created. This is because pyplot does not support generating GIF output files on Windows. The PNG file is then converted to GIF by creating an object of the Image class from the Pillow library.

9.14.3.1.5 IV_Swinger2_plotter run() method

The *run()* method overrides the base class *run()* method, but is very different. It does the following:

- Adds sensor info (if any) to the curve names by calling the *add_sensor_info_to_curve_names*() method
- Creates an argparse.Namespace object and calls the *set_default_args()* method to populate it with the default values
- Changes to the plot directory so files will be created there
- Creates an IV_Swinger_extended object and sets its properties to the defaults
- Creates a CsvFileProcessor object to process all CSV files
- If the *generate_pdf* property is set, calls the *plot_graphs_to_pdf()* method
- Calls the *plot_graphs_to_gif()* method
- Captures the maximum X and maximum Y plot dimensions (for locking)
- Logs the MPP values

9.14.3.2 IV_Swinger2 Class *plot_results()* Method

The IV_Swinger2 class's *plot_results()* method does the following:

- Creates an IV_Swinger2_plotter object
- Copies the relevant IV_Swinger2 properties to the plotter object's properties
- Calls the plotter object's run() method
- Copies the plotter's resulting *current_img*, *max_x* and *max_y* properties to the IV_Swinger2 properties

The *plot_results()* method is called by the *swing_curve()* method after it processes the ADC values. It also can be called by the instantiator of an IV_Swinger2 object (e.g. the GUI) to generate a new image from one or more existing data points CSV files.

9.15 Calibration Support

Calibration is supported for the following:

- Vref (+5V)
- Voltage and Current
- Resistors
- Pyranometer

This section covers the calibration support in the IV_Swinger2.py module; it does not cover the GUI interface for performing the calibrations, but does mention what the GUI (or other higher level code) must do to effect a calibration.

9.15.1 Calibration Configuration and Properties

The Configuration base class has a [Calibration] section containing all of the calibration values, which are saved to and restored from the configuration file. Each value corresponds to a property of the IV_Swinger2 class.

A calibration is performed (at a higher level, such as the GUI) by the following:

- Updating the IV_Swinger2 object property
- Updating the configuration value
- Saving the updated configuration to the configuration file

9.15.2 Arduino EEPROM Storage of Calibration Values

The calibration values for the following are stored in the Arduino EEPROM:

- Voltage and Current
- Resistors

They are also stored in the configuration file. The first time the IV Swinger 2 hardware is used, it will have an empty EEPROM, so the default values from the configuration are written to the EEPROM. From then on, the configuration values are always overridden by the values read from EEPROM. These actions are performed in the *wait_for_arduino_ready_and_ack()* method of the IV_Swinger2 class.

When a calibration is performed, the EEPROM values need to be updated. This is accomplished (at a higher level, such as the GUI) by the following:

- Updating the configuration value(s)
- Resetting the Arduino
- Calling the *wait_for_arduino_ready_and_ack()* method with *write_eeprom*=True

As of software release v2.6.0, one additional calibration value is stored in the Arduino EEPROM:

- Arduino ATmega328 microcontroller bandgap voltage (in microvolts)

Unlike the others, this value is not stored in the configuration file. It is determined by calling the *calibrate_bandgap()* method, which updates the *bandgap_microvolts* property. This property's value is written to EEPROM by resetting the Arduino and calling the *wait_for_arduino_ready_and_ack()* method with *write_eeprom*=True. See the next section for more information.

9.15.3 Vref Calibration

The nominal value of the MCP3202 reference voltage (Vref) is +5 V. A more accurate calibrated value may be used by overriding the default for the following:

- IV_Swinger2 property: *adc_vref*
- [Calibration] config item: *vref*
- Default: 5.0

The *adc_vref* property is used by the *adc_inc* property to determine the voltage of one ADC increment. The *adc_inc* property is used by the *v_mult* and *i_mult* properties, among others.

In software release v2.6.0, a new feature was added that uses the Arduino 1.1 V bandgap reference to measure Vref for every IV curve. This feature effectively replaces Vref calibration with bandgap calibration (for backward compatibility and to avoid user confusion, it is still referred to as Vref calibration, however). The *adc_vref* property is used as before, and the higher-level (e.g. GUI) code could still choose to use it as the calibrated quantity, performing the steps in section 9.15.1 above. But that would not utilize the new feature.

The IV_Swinger2 class now provides a *calibrate_bandgap()* method that the higher-level code calls immediately after it sets the *adc_vref* property to the user-measured value during a "Vref calibration". The *calibrate_bandgap()* method calls the *read_bandgap()* method, which sends a READ_BANDGAP message to the Arduino sketch and parses its response using the *parse_bandgap_msg()* method. The *calibrate_bandgap()* method then uses the following equation to determine the bandgap voltage:

$$V_{bandgap} = \frac{ADC}{1024} \cdot V_{ref}$$

This is the calibrated bandgap voltage that is written to the Arduino EEPROM. The bandgap voltage is very stable for a given ATmega328 even at different temperatures and over time. So, once it is known, it can be used to measure Vref using a rearrangement of the equation above.

9.15.4 Voltage and Current Calibration

Voltage and current calibration are performed (at a higher level, such as the GUI) by overriding the defaults for the following:

- IV_Swinger2 properties: *v_cal, v_cal_b, i_cal, i_cal_b*
- [Calibration] config items: *voltage, voltage intercept, current, current intercept*
- Defaults: 1.0, 0.0, 1.0, 0.0

These adjustments are applied to the raw ADC values by the *calibrate_adc_pairs()* method, before the Vref and resistors adjustments. The first value is the slope of the linear calibration function. The second value is the y-intercept of the linear calibration function.

A basic (1-point) calibration updates the *v_cal* or *i_cal* value only. This is just a simple scaling calibration. An advanced (2-point) calibration updates both *v_cal* and *v_cal_b* or *i_cal* and *i_cal_b*. The calibration uses the familiar "y = mx + b" equation for a line:

calibrated CH0 ADC value = (raw CH0 ADC value) * *v_cal* + *v_cal_b*
calibrated CH1 ADC value = (raw CH1 ADC value) * *i_cal* + *i_cal_b*

9.15.4.1 Advanced Calibration Support

The following IV_Swinger2 methods are provided to support the advanced voltage and current calibration features:

- *request_adv_calibration_vals()*

  Requests an IV curve from the Arduino for the purpose of getting the Voc or Isc values for an advanced calibration. This is used for the advanced voltage calibration (EMR or SSR) and for the EMR current calibration. It is not used for the SSR current calibration.

- *get_adv_voltage_cal_adc_val()*

  Called after *request_adv_calibration_vals()* to extract the ADC value of the V_{OC} point into the *adv_cal_adc_val* property.

- *get_adv_voltage_cal_volts()*

  Converts the value of the *adv_cal_adc_val* property to volts (uncalibrated).

- *get_emr_adv_current_cal_adc_val()*

  Called after *request_adv_calibration_vals()* to extract the ADC value of the I_{SC} point into the *adv_cal_adc_val* property.

- *get_adv_current_cal_amps()*

  Converts the value of the *adv_cal_adc_val* property to amps (uncalibrated).

- *request_ssr_adv_current_calibration_val()*

  Sends the Arduino a DO_SSR_CURR_CAL config message that tells it to configure the SSRs to pass current around the load capacitors and the bleed resistor (if there is one) for three seconds. The ADC value for the channel measuring current in this state is assigned to the *adv_cal_adc_val* property. This is used for SSR advanced current calibration.

These methods are provided for higher level code (such as the GUI) to get the uncalibrated voltage or current measurements for each of the 2 points for an advanced calibration using the following call sequences to the above methods:

Voltage (EMR or SSR):

request_adv_calibration_vals()
get_adv_voltage_cal_adc_val()
get_adv_voltage_cal_volts()

Current (EMR):

request_adv_calibration_vals()
get_emr_adv_current_cal_adc_val()
get_adv_current_cal_amps()

Current (SSR):

request_ssr_adv_current_calibration_val()
get_adv_current_cal_amps()

9.15.5 Resistors Calibration

The measured resistances of the resistors can be specified (at a higher level, such as the GUI) by overriding the defaults for the following:

- IV_Swinger2 properties: *vdiv_r1, vdiv_ r2, amm_op_amp_rf, amm_op_amp_rg, amm_shunt_max_volts*
- [Calibration] config items: *r1 ohms, r2 ohms, rf ohms, rg ohms, shunt max volts*
- Defaults: 150000, 7500, 75000, 1000, 5000

For "legacy" (IVS1) reasons, the shunt resistor is specified by two values: max volts and max amps. Its resistance is max_volts/max_amps. The max_amps value is hardcoded to 10A, so we just keep the value of shunt max volts in the config.

The *vdiv_r1* and *vdiv_ r2* properties are used to determine the *vdiv_ratio* property, which in turn is used to determine the *v_mult* property. The *amm_op_amp_rf, amm_op_amp_rg,* and *amm_shunt_max_volts* properties are used to determine the *amm_op_amp_gain* property (IV_Swinger base class), which in turn is used to determine the *i_mult* property. The *amm_shunt_max_volts* property (IV_Swinger base class) is used to determine the *amm_shunt_resistance* property, which in turn is also used to determine the *i_mult* property.

9.15.6 Pyranometer Calibration

Pyranometer calibration is performed (at a higher level, such as the GUI) by overriding the defaults for the following:

- IV_Swinger2 properties: *pyrano_cal, pyrano_cal_a, photodiode_pct_per_deg_c*
- [Calibration] config items: *pyranometer, pyranometer a coeff, pyranometer pct per degc*
- Defaults: 4.3, 0.0, 0.16

These properties are used by the *convert_ads1115_val_to_w_per_m_squared()* method.

9.16 Other IV_Swinger2 Class Capabilities

So far in this chapter, the following primary capabilities of the IV_Swinger2 class have been discussed:

- Arduino interaction
- Swinging an IV curve
- Plotting results
- Calibration

There are several other secondary capabilities that the class provides:

- Bias battery support
- Optional sensor support
- Clean-up support

These will not be discussed in detail, but their relevant methods will be listed.

9.16.1 Bias Battery Support

The PV cell versions of the hardware may require a bias battery, as described in Section 7.2.4.1 on page 72 and Section 7.2.4.2 on page 75. The following methods are provided to support the use of a bias battery:

- *swing_battery_calibration_curve()*
- *gen_bias_batt_adc_csv()*
- *apply_battery_bias()*
- *get_bias_batt_csv()*
- *remove_prev_bias_battery_csv()*

9.16.2 Optional Sensor Support

The following methods are provided to support the optional irradiance and temperature sensors:

- *create_run_info_file()*
- *write_sensor_info_to_file()*
- *translate_ads1115_msg_to_photodiode_temp_scaling()*
- *convert_ads1115_val_to_deg_c()*
- *translate_ads1115_msg_to_irradiance()*
- *convert_ads1115_val_to_w_per_m_squared()*
- *update_irradiance()*

9.16.3 Clean-up Support

The following methods are used to remove unneeded files:

- *clean_up_after_failure()*
- *clean_up_files()*

- *clean_up_file()*

9.17 IV_Swinger2.py *main()* Function

The IV_Swinger2.py module has a *main()* function at the end of the file. It is not used unless the module is run standalone, rather than being imported. Its purpose is more to be an example than to really be useful.

The *main()* function swings one IV curve, stores the result in the standard (OS-dependent) place, and opens the PDF. The configuration is read from the standard place. A copy of the configuration is saved in the run directory, but the original config is restored.

9.18 Graphical User Interface

The IV_Swinger2_gui.py module provides a tkinter/tkinter.ttk GUI for IV Swinger 2. It is dependent on the IV_Swinger.py, IV_Swinger_plotter.py and IV_Swinger2.py modules. The opposite is not true, however; the lower-level modules can be used standalone or could potentially be used by an alternate user interface module.

Figure 9-2 on page 121 is useful to see the relationships between the GUI classes and the lower-level classes.

This section describes the basic operation of the IV_Swinger2_gui.py module. It assumes that the reader understands GUIs and GUI terminology in general and has knowledge of tkinter and tkinter.ttk. While tkinter and tkinter.ttk are not particularly well documented, virtually every question I had was already asked and answered (mostly by Bryan Oakley) on Stack Overflow.

9.18.1 IV_Swinger2_gui.py *main()* Function

The IV_Swinger2_gui.py module is intended to be run from the command line or invoked by clicking an icon. In either case, its *main()* function is called.

The *main()* function does just two things:

- Creates a GraphicalUserInterface object
- Calls that object's *run()* method

Everything else follows from that.

It does have an exception handler that is described at the end of Section 9.18.7 on page 170.

9.18.2 GraphicalUserInterface Class

The GraphicalUserInterface class is derived from the tkinter.ttk.Frame class.

9.18.2.1 Initialization

The GraphicalUserInterface class's *__init__()* method starts by creating a root Tk object and setting some of its options with the *set_root_options()* method before passing it to the base class's *__init__()* method. The root object is the main window of the application. It is filled by the GraphicalUserInterface frame and is also the "engine" of the GUI, performing the event handling, scheduling, etc.

Next in *__init__()*, an object of the IV_Swinger2 class (from the IV_Swinger2.py module) is created. This object is named "ivs2" and provides all of the services described in the earlier sections of this chapter (Arduino communication, swinging IV curves, plotting, etc.)

After calling the *get_app_dir()*, *get_version()* and *set_grid()* methods, the *__init__()* method creates a Configuration object. The Configuration class is an extension of the Configuration class that is defined in the IV_Swinger2.py module. The Configuration object's *get()* method is called to update itself with the values read from the config file.

After calling the *start_to_right()* and *set_style()* methods, the *__init__()* method calls the *create_menu_bar()* and *create_widgets()* methods. The *create_menu_bar()* method creates a MenuBar object. The *create_widgets()* method calls other methods to create all of the labels, buttons, check boxes, etc. on the main window of the application.

Finally, the *__init__()* method logs some debug information and starts the USB monitor, which checks for USB cable disconnection.

9.18.2.2 *GraphicalUserInterface run()* method

The creation/initialization of the GraphicalUserInterface object results in a configured but inactive GUI. Its *run()* method gets it started by calling the *mainloop()* method of the root Tk object. This is a blocking method call; it blocks until the root window is closed.

Before it calls the *mainloop()* method, however, the *run()* method schedules a call to the *attempt_arduino_handshake()* method. This is a non-blocking method call (using tkinter's *after()* method for scheduling). There is no guarantee that the hardware is even connected when the application is started, but if it is, this hides the latency of connecting to it so the user sees that it is connected as soon as the GUI comes up.

Also before the call to *mainloop()*, the *start_on_top()* method is called. That method just configures the root window to open on top of existing windows rather than underneath them. A callback to the *close_gui()* method is also registered for when the root window's close button is clicked. The *close_gui()* method does some clean-up before calling the *destroy()* method of the root object. It also calls the configuration object's *log_cfg_diffs()* method, which records the differences in the configuration at the time of exit versus the starting configuration.

9.18.3 Menu Bar

The MenuBar class is derived from the tkinter Menu class. The menu items are also this same class. This is standard for a tkinter application. For the most part, the menu bar works as expected on different platforms without doing anything special. On a Mac, the menu bar is displayed at the top of the screen. On a Windows machine, it is displayed at the top of the application's window.

Each menu on the menu bar is created with a *create_xxx_menu()* method, e.g. *create_calibrate_menu()*. Each of these methods is called in the order of its position on the menu bar (left to right). These methods:

- Create a Menu object
- Add that object to the menu bar using the *add_cascade()* method of the menu bar object
- Add commands to the menu using the *add_command()* method of the menu object

The *add_command()* method registers a callback method to be executed when the command is selected from the menu. For example, the "View Log File" command on the "File" menu registers a callback to the *view_log_file()* method which brings up the dialog to select and open the log file.

A "postcommand" callback method is registered for some menus when the Menu object is created. This method is run immediately after the menu is created, and is used to update the menu based on current information. In some cases, this is to disable ("gray out") menu commands that are not valid. In the case of the USB Port menu, it populates the menu with the current list of connected USB ports.

9.18.4 Main Window Widgets

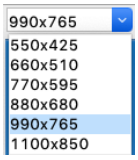
The widgets added to the main window by the *create_widgets()* method are defined by classes derived from the following standard tkinter.ttk widget classes[16]:

- tkinter.ttk.Label
- tkinter.ttk.Combobox
- tkinter.ttk.Button
- tkinter.ttk.Entry
- tkinter.ttk.Checkbutton

The tkinter.ttk.Label widgets display information, but accept no input from the user.

The tkinter.ttk.Combobox, tkinter.ttk.Button, tkinter.ttk.Entry and tkinter.ttk.Checkbutton widgets all cause an action when the user interacts with them. Generally, this means invoking a callback method that was registered when the widget was created.

9.18.4.1 Image Size Combobox



[16] Note that most of these hyperlinks are for the tkinter version of the widget. The tkinter.ttk version is the same other than the removal of options related to styling such as "fg" and "bg".

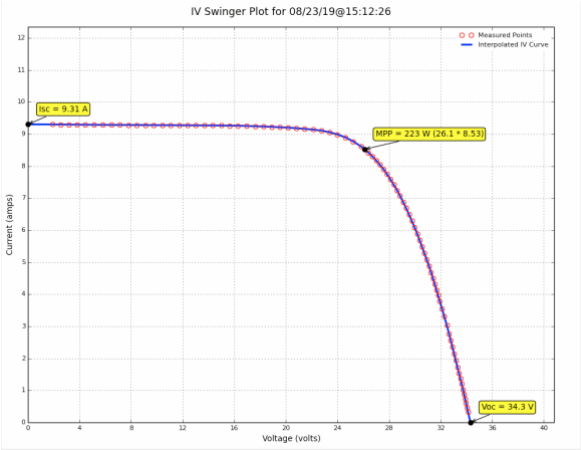
The image size combobox invokes the *update_img_size()* callback method when the image size is selected. Since the image size affects the other widgets too, this method is a bit more complex than one might imagine, but the details will be left to the code and comments.

9.18.4.2 Version Label

Version: v2.5.1

The version label is a simple tkinter.ttk.Label object showing the application version string which was read from the version.txt file during initialization.

9.18.4.3 Image Pane



The image pane widget (ImagePane class) is actually a tkinter.ttk.Label, which seems like a misnomer. But that is the best way to display an image in tkinter. Three types of image can be displayed in the image pane:

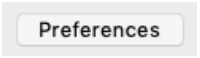
- The splash image that comes up when the app is started
- The image containing an IV curve or overlay of multiple IV curves
- An error message image (used for errors while looping)

The ImagePane class has a *display_img()* method that uses the Image and ImageTk classes from the Pillow library to open an image file and create an object to assign to the "image" option of the tkinter.ttk.Label object. This method is used only for displaying the splash and error messages. It is called by the ImagePane class's *display_splash_img()* and *display_error_img()* methods. It scales those images to fit the current image size dimensions specified by the image size combobox. This scaling makes it possible to use fixed size Splash_Screen.png and Blank_Screen.png files that work regardless of the values used in the image size combobox, including values not in the combo list.

The GraphicalUserInterface class also has a *display_img()* method, which is used to display IV curves and overlays. It changes the *image* attribute and "image" option of the ImagePane object to a tkinter PhotoImage object loaded from the specified image file. This method does not use the Pillow library, and does not do any scaling, so the image is displayed at its native size regardless of the current values used in the image size combobox. In most cases there would be no discrepancy, but when using the

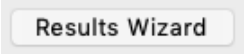
Results Wizard to look at old runs or overlays, there may be some that are different sizes. Scaling those would not only slow things down, but would also result in scaling artifacts in the displayed image.

9.18.4.4 Preferences Button



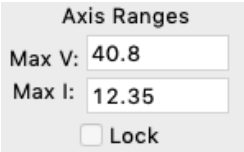
The Preferences button invokes the *show_preferences()* callback method, which brings up the Preferences dialog by creating a PreferencesDialog object.

9.18.4.5 Results Wizard Button



The Results Wizard button invokes the *results_actions()* callback method, which brings up the Results Wizard dialog by creating a ResultsWizard object.

9.18.4.6 Axis Range Entry Boxes and Lock Checkbutton



The axis range entry boxes are objects of the tkinter.ttk.Entry class. The *apply_new_ranges()* callback method is invoked when the user hits Return after entering a value. That method updates the range properties and calls the *redisplay_img()* method to use the lower level modules' plotting support to recreate and display the whole image with the new axis range.

9.18.4.7 Go Button

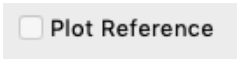
The Go button (i.e. the "Swing! Button) is an object of the GoStopButton class, which is derived from the tkinter.ttk.Button class. It invokes the *go_actions()* callback method, which calls the *swing_loop()* method to swing an IV curve, possibly looping.

9.18.4.8 Plot Power Checkbutton

The Plot Power checkbutton is an object of the PlotPower class, which is derived from tkinter.ttk.Checkbutton. When its value is changed, the PlotPower *update_plot_power()* callback method is called, which calls the *handle_plot_power_or_ref_event()* method to update the IV_Swinger2 object's

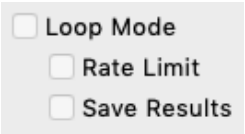
plot_power property, update the configuration, and redisplay the current image (with power plotted or not plotted, as specified).

9.18.4.9 Plot Reference Checkbutton



The Plot Reference checkbutton is an object of the PlotRef class, which is derived from tkinter.ttk.Checkbutton. When its value is changed, the PlotRef *update_plot_ref()* callback method is called, which calls the *handle_plot_power_or_ref_event()* method to update the IV_Swinger2 object's plot_ref property, update the configuration, and redisplay the current image (with the PV model reference curve plotted or not plotted, as specified).

9.18.4.10 Looping Control Checkbuttons



The looping control checkbuttons are objects of the following classes, derived from tkinter.ttk.Checkbutton:

- LoopMode
- LoopRateLimit
- LoopSaveResults

Refer to the code and comments for details. Section 9.18.5.1 below describes loop mode.

9.18.4.11 Tooltips

Tooltips are implemented for most of the widgets on the main window. They are also implemented for many of the ResultsWizard widgets.

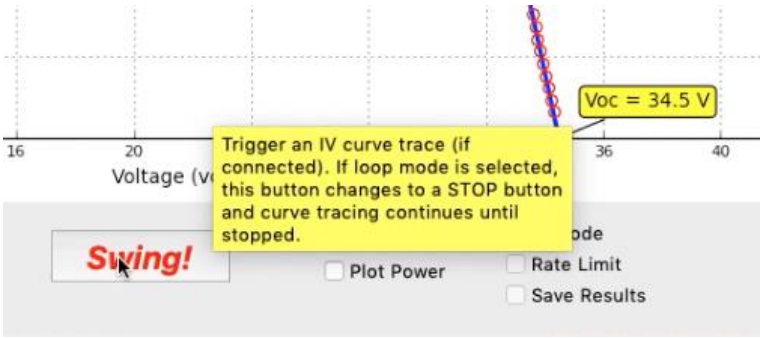


Figure 9-6: Go Button Tooltip

The Tooltips.py module is a modified version of code posted in a Stack Overflow thread. When the widget object is created, a Tooltip object is created and bound to the widget. One of the modifications

was to make the tooltips disappear after an amount of time that is proportional to the number of characters in the text. That and the other characteristics of the tooltips were implemented with the objective that the tooltips be useful for novice users but not annoying for experienced users. There is no support for disabling the tooltips.

9.18.5 Swinging IV Curves

The *swing_loop()* method is called by the *go_actions()* callback method when the Go button is pressed. This method invokes the IV_Swinger2 object's *swing_curve()* method to swing the IV curve, and then it displays the generated GIF in the image pane using the GraphicalUserInterface *display_img()* method. The configuration file is saved in the run directory and temporary files are cleaned up. That's all, in the simple case where loop mode is not enabled and dynamic bias battery calibration is not enabled.

9.18.5.1 Loop Mode

If the Loop Mode checkbutton is checked, the *swing_loop()* method calls the *add_stop_button()* method to create a Stop button from the GoStopButton class before calling the *swing_curve()* method. The Stop button obscures the Go button. After the *swing_curve()* method is called, *swing_loop()* ends by scheduling another call of itself after the programmed delay. In that sense it appears to be a loop. Unlike an actual loop, however, it is non-blocking. This is essential in order for the GUI not to lock up. The *first_loop* argument to the *swing_loop()* method is set to False so the method knows not to add a new Stop button on the non-first iterations. The *stop_actions()* callback method is invoked when the Stop button is pressed, which stops the looping and removes the Stop button, exposing the Go button underneath it. The looping is stopped by canceling the next iteration's scheduled call to *swing_loop()*.

If the "Stop on non-fatal errors when looping" checkbutton is not checked on the Looping tab of the Preferences dialog, the normal error handling is overridden. Normally, when any error is detected, an error dialog is displayed and the looping is stopped. But when that checkbutton is unchecked, and an I_{SC}=0, V_{OC}=0 or I_{SC} Stable Timeout error is detected, the *display_screen_err_msg()* method is called and looping continues after cleaning up the failed run directory. The *display_screen_err_msg()* method displays the error message as an image in the image pane. This is so the user can still see the error message but does not have to close a dialog.

9.18.5.2 Dynamic Bias Battery Calibration Curve

More complexity comes in if a bias battery is being used and dynamic bias battery calibration is enabled. In that case, the *swing_loop()* method calls the IV_Swinger2 object's *swing_battery_calibration_curve()* method before calling the *swing_curve()* method. That method calls the *swing_curve()* method itself, so for each press of the Go button (or each loop), two IV curves are swung: first one for the bias battery calibration and then one for the series connection of the bias battery and the PV cell. The *gen_bias_batt_adc_csv()*, *remove_prev_bias_battery_csv()* and *copy_file_to_parent()* methods are also called and the second relay is turned on by setting the *second_relay_state* property.

9.18.6 Dialogs

Dialogs are child windows of the main application window. They range from very simple information, warning and error dialogs to complex Preferences and Result Wizard dialogs.

A dialog can be modal or modeless. A modal dialog is easier to deal with from a programming point of view because there is no concern about actions in the main window conflicting with the dialog window. However, this can prevent some very useful functionality. All but one of the IV Swinger 2 dialogs are modal since there is no reasonable use case for making them modeless. The exception is the Results Wizard dialog.

9.18.6.1 tkinter Dialogs

tkinter provides some generic dialogs that can be used as-is:

- tkinter.tkmessagebox
- tkinter.tksimpledialog
- tkinter.tkfiledialog

9.18.6.1.1 tkinter.tkmessagebox

The tkinter.tkmessagebox module is used for simple information, warning and error messages (showinfo, showwarning, showerror). These have no user interaction other than the OK button to dismiss the dialog. The only other tkmessagebox dialog that is used is the "askyesno" dialog, which gives the user two buttons: Yes and No. The vast majority of tkmessagebox dialogs are used for error messages (showerror), with over 70 such dialogs. There are 10 or fewer of each of the other three types.

All tkmessagebox dialogs are modal.

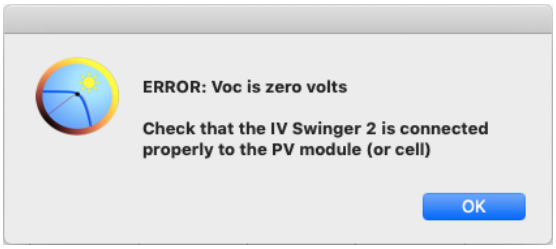


Figure 9-7: tkmessagebox dialog (showerror)

9.18.6.1.2 tkinter.tksimpledialog

A modified tkinter.tksimpledialog module is used for dialogs where all that is needed from the user is a string, integer or floating point value (askstring, askinteger, askfloat). It has built-in checking of the user-entered value to make sure it is the required type.

The tkinter.tksimpledialog module is slightly modified in the MyTkSimpleDialog.py file. The changes are:

- Fix a problem where the OK and Cancel buttons are sometimes invisible unless/until the dialog is resized
- Swap the positions of OK and Cancel when run on Mac

All MyTkSimpleDialog dialogs are modal.

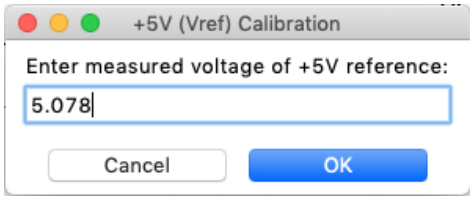


Figure 9-8: MyTkSimpleDialog (askfloat)

9.18.6.1.3 tkinter.tkfiledialog

The tkinter.tkfiledialog module is used for dialogs that require the user to select a folder/directory or file on their computer. Its appearance and structure are platform-specific and it uses the interface that is typical for such dialogs on the platform: on MacOS it is a Finder dialog; on Windows it is a File Explorer dialog. This is all taken care of by tkinter.

All tkfiledialog dialogs are modal.

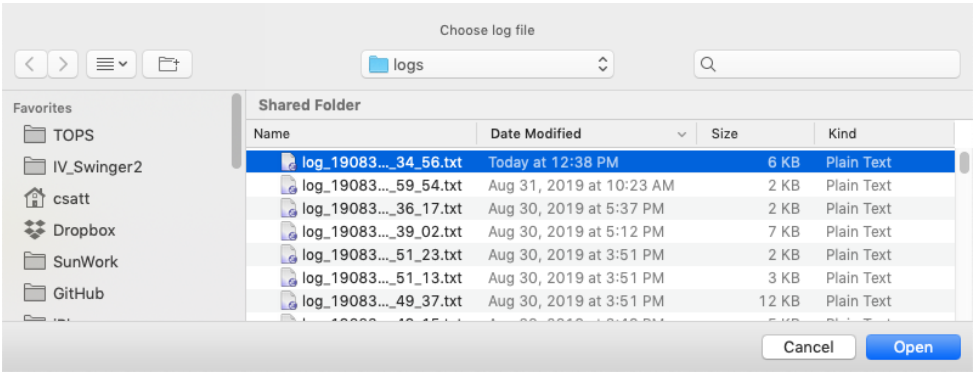


Figure 9-9: tkfiledialog (Mac)

9.18.6.2 Dialog Base Class

Dialogs that cannot be implemented using the built-in tkinter dialogs are created using classes defined in the IV_Swinger2_gui.py module. Other than the Results Wizard, these all are subclasses derived from the Dialog base class.

The Dialog base class evolved from the example on effbot.org and is derived from the tkinter Toplevel widget class. It is a modal dialog with OK and Cancel buttons that are positioned depending on what is customary for the platform. Placeholder methods to create the body and perform the appropriate actions when the OK or Cancel button is pressed are provided for the subclass to override. A placeholder function to validate the input before applying it is also provided for optional override. The cancel behavior was enhanced to support reverting to snapshot values using placeholder *snapshot()* and *revert()* methods.

Initialization parameters customize the following:

- Dialog window title
- Exclusion of OK and/or Cancel button
- Whether the Return key equates to clicking OK

- Whether the dialog is resizable
- Minimum and maximum window height

9.18.6.3 Dialog Geometery

The IV_Swinger2_gui class has a method called *set_dialog_geometry()* that can be used on any dialog that is derived from the tkinter Toplevel widget class. It sets the initial size and position of the dialog window and constrains its resizing.

The tkinter geometry manager initially sizes the dialog window to fit its contents and places it somewhere to its liking. The *set_dialog_geometry()* method uses that size as a starting point, but may override the height and always overrides the position. These modifications occur before the window appears.

The *set_dialog_geometry()* method has *min_height* and *max_height* parameters that are used for resizable dialogs. If *min_height* is specified, the window will be sized to that value initially. The *max_height* parameter is only relevant if *min_height* is specified; if *max_height* is not specified, the height when the method was called will be used as the maximum. If *min_height* is not specified, the height is not changed. The *set_dialog_geometry()* method locks the width of the dialog window so that resizability is only in the height dimension.

The *set_dialog_geometry()* method chooses where to put the dialog window based on how much space exists on the screen to the left and right of the root (main) window. If there are enough screen pixels to the left of the root window, then it is put it there (with 10 pixels of overlap). The second choice is to the right of the root window. The last choice is whichever side has more space, overlapping the root window by as much as necessary to leave 10 pixels of screen.

9.18.6.4 Help Dialogs

Help dialogs are derived from the Dialog base class, but use few of its features. They have no Cancel button and none of the base class methods except *body()* is overridden, so the OK button just closes the dialog. Help dialogs are all resizable (height only), with a *min_height* of 360 pixels and a *max_height* of 2000 pixels.

The *body()* method of help dialogs creates a tkinter.scrolledtext widget. Its height is initially 1 pixel, but by using the pack geometry manager with fill=BOTH and expand=True, it expands to fill the window. Text is inserted in the scrolledtext widget with its *insert()* method after configuring fonts with its *tag_configure()* method.

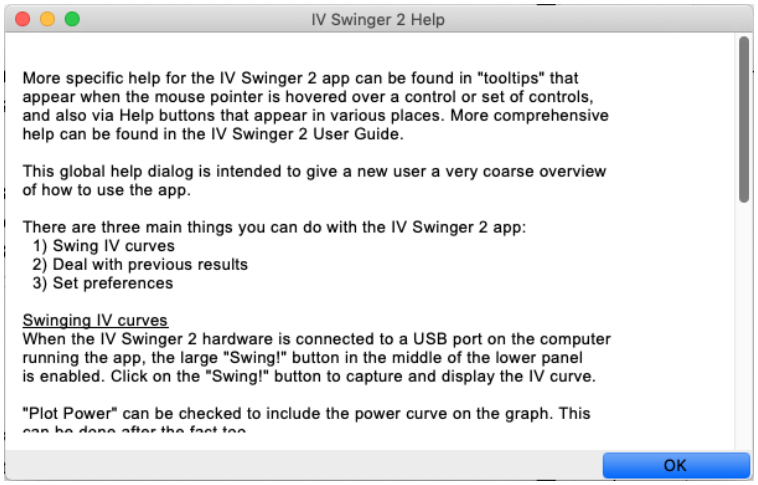


Figure 9-10: Help Dialog

9.18.6.5 Calibration Dialogs

The simpler calibration dialogs use MyTkSimpleDialog, but the following more complex calibration dialogs are derived from the Dialog base class:

- ResistorValuesDialog
- BiasBatteryDialog
- AdvCalDialog (with subclasses AdvCurrentCalDialog and AdvVoltageCalDialog)

These dialogs use many of the features of the base class by overriding its placeholder methods.

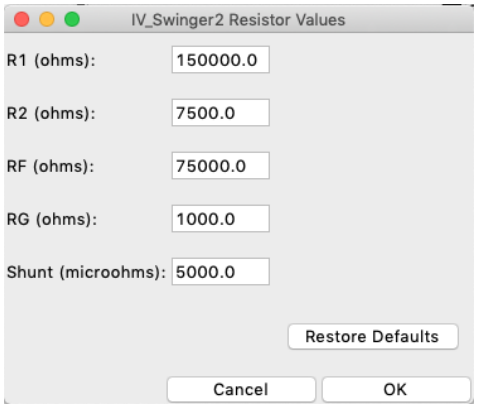


Figure 9-11: ResistorValuesDialog

All calibration dialogs interact with the main window's IV_Swinger2 object to change its calibration properties and configuration values based on user input. Details are documented in the comments and code.

9.18.6.6 Preferences Dialog

The most complex dialog that is derived from the Dialog base class is the PreferencesDialog class.

The *body()* method of the PreferencesDialog class creates a tkinter.ttk.Notebook widget object to which it adds the four tabs (Plotting, Looping, Arduino and PV Model), each of which is a tkinter.ttk.Frame object. Each of the four tabs is then populated with all of its respective widgets.

Note that the OK and Cancel button widgets are not part of the dialog body. This means their actions are independent of which tab is active.

9.18.6.6.1 Plotting Tab

The Plotting tab frame uses the following widget types:

- tkinter.ttk.Label
- tkinter.ttk.Radiobutton
- tkinter.ttk.Scale
- tkinter.ttk.Entry
- tkinter.ttk.Button

Except for the labels and buttons, each of these widgets controls a tkinter StringVar() variable that maps to a configuration and/or property value. When the widget is used, the *immediate_apply()* method is called so its effects can be seen immediately on the current plot in the image pane.

The Font name List button generates a FontListDialog dialog, which is a lot like a help dialog, but contains a list of available fonts generated by the IV_Swinger class's *get_and_log_pyplot_font_names()* method. That method gets the list of fonts from matplotlib.fontmanager. The user has to copy/paste from this list to the Font name entry box; there is no support for just clicking on a font name to select it.

The Help button generates a PlottingHelp dialog, which is a typical help dialog.

The Restore Defaults button calls the *restore_plotting_defaults()* method, which returns all widgets' tkinter StringVar() variables to their default values. It then calls the *immediate_apply()* method.

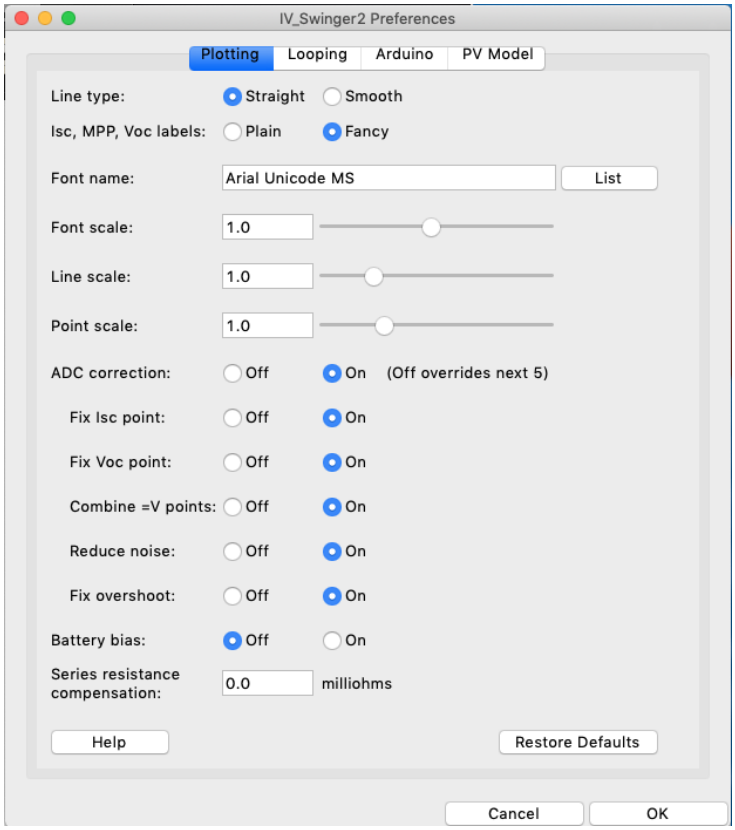


Figure 9-12: Preferences Dialog Plotting Tab

9.18.6.6.2 Looping Tab

The Looping tab frame has two tkinter.ttk.Checkbutton widgets and a tkinter.ttk.Button help button only. The checkbuttons control their respective tkinter StringVar() variables, which are mapped to configuration and property values.

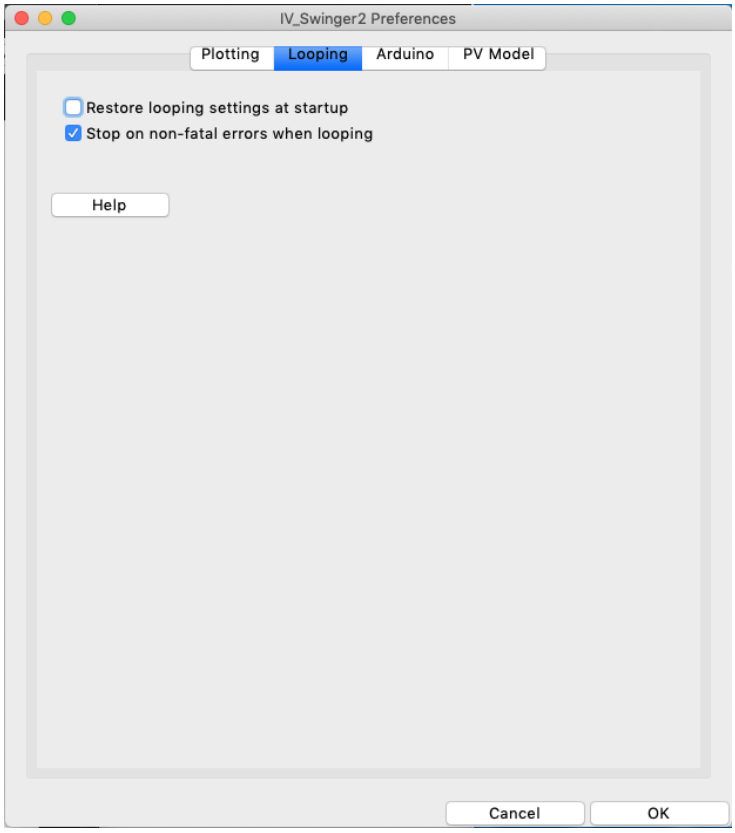


Figure 9-13: Preferences Dialog Looping Tab

9.18.6.6.3 Arduino Tab

The Arduino tab frame uses the following widget types:

- tkinter.ttk.Label
- SpiClkCombo (derived from tkinter.ttk.Combobox)
- tkinter.ttk.Entry
- tkinter.ttk.Checkbutton
- tkinter.ttk.Button

Except for the labels and buttons, each of these widgets controls a tkinter StringVar() variable that maps to a configuration and/or property value.

The Help button generates an ArduinoHelp dialog, which is a typical help dialog.

The Restore Defaults button calls the *restore_arduino_defaults()* method, which returns all widgets' tkinter StringVar() variables to their default values.

Note that, unlike the Plotting tab, changing values on the Arduino tab does not affect the current plot in the image pane. Changes will affect future IV curves, but it is too late to affect past curves.

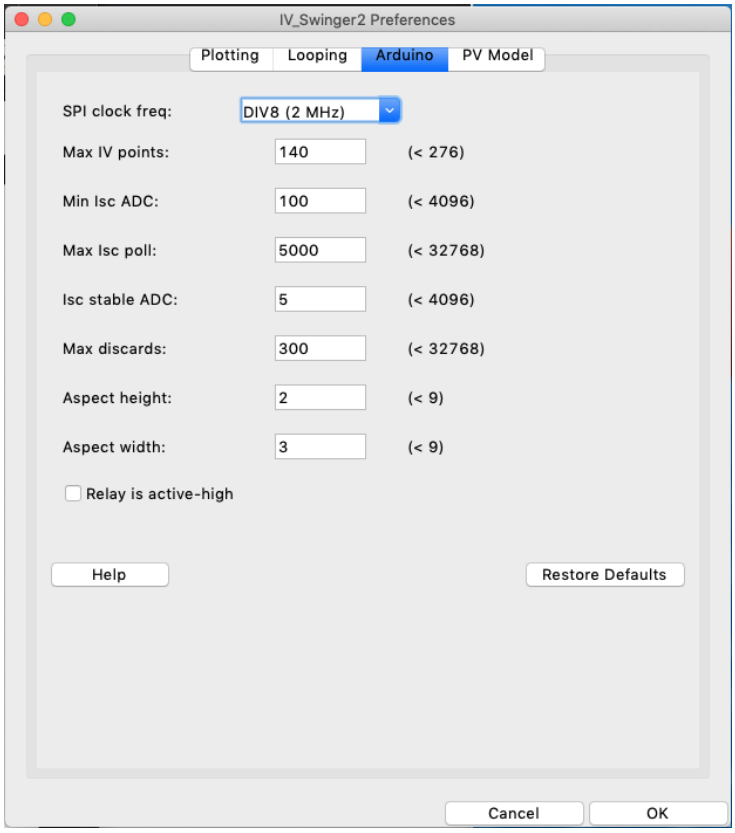


Figure 9-14: Preferences Dialog Arduino Tab

9.18.6.6.4 PV Model Tab

The PV Model tab frame uses the following widget types:

- tkinter.Listbox
- tkinter.ttk.Scrollbar
- tkinter.ttk.Label
- tkinter.ttk.Entry
- tkinter.ttk.Button
- tkinter.ttk.Combobox
- tkinter.ttk.Separator
- tkinter.ttk.Checkbutton

The PV Model tab is more complex than the others. Like the other tabs, there are configuration and property values that are controlled by the PV Model tab. However, these are limited to the name of the PV currently being tested and the values of the controls near the bottom of the tab (below the separator line). The PV characteristics are stored separately from the configuration in the PV spec CSV file. The upper part of the tab may be used to edit the contents of the PV spec CSV file, i.e. adding / deleting / modifying entries in that file, which may contain the specifications for many PVs. In addition to being the interface for the user to specify the characteristics of the PV(s) being tested, it supports testing the ability of the software to model a PV with those characteristics at different irradiances and cell temperatures.

The tkinter.Listbox widget contains the list of PV names from the PV spec CSV file. It allows the user to select an entry from that list to use for plotting PV reference curves or for editing and testing. Since this list may be too big to fit in the Listbox, a Scrollbar is paired with it. When the PreferencesDialog object is created, it calls its *initialize_pv_specs()* method to populate its *pv_specs* list from the PV spec CSV file. If the file does not exist, it creates one by calling the *create_pv_spec_file()* function from IV_Swinger_PV_model.py. The *pv_specs* list is used to populate the entries in the listbox.

Except for the labels and buttons, each of these widgets controls a tkinter StringVar() variable that maps to a configuration and/or property value.

The Help button generates an PvModelHelp dialog, which is a typical help dialog.

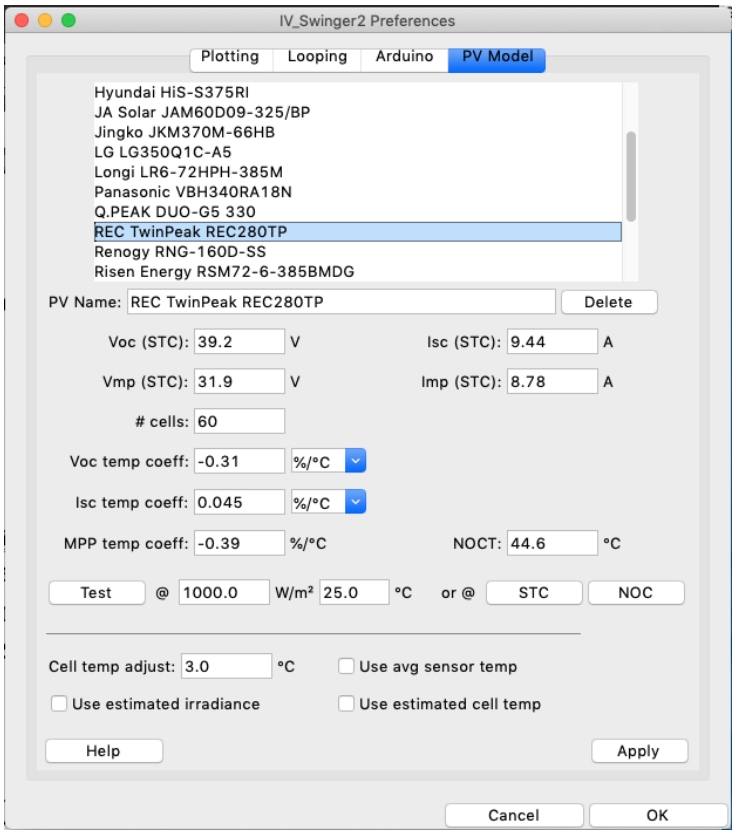


Figure 9-15: Preferences Dialog PV Model Tab

9.18.6.6.5 validate() Method

The PreferencesDialog class's *validate()* method overrides the placeholder method of the Dialog base class. It performs validity checks on all of the tkinter StringVar() variables associated with the widgets on the Plotting and Arduino tabs. If any check fails, it displays an error dialog describing the problem and returns a value of False. If all checks pass, it returns True.

9.18.6.6.6 apply() Method

The PreferencesDialog class's *apply()* method overrides the placeholder method of the Dialog base class. As defined in the base class, it is called by the *ok()* callback method when the OK button is pressed, just before closing the dialog. In this case, it first calls the *validate()* method. If that returns False (validation failed), the *apply()* method returns without taking any actions. If the *validate()* method returns True (validation passed), the *apply()* method calls the *plotting_apply()*, *looping_apply()* and *arduino_apply()* methods which set the associated configuration and property values based on the tkinter StringVar() variable values.

9.18.6.6.7 immediate_apply() Method

The *immediate_apply()* method is only relevant to the Plotting tab. Its purpose is to call the *apply()* method without the OK button being pressed. It is called by the widgets on the Plotting tab when the widget is used to change a value. This allows the user to immediately see the effect of the change.

This adds some complexity, however, because the user should still be able to use the Cancel button to exit the dialog without committing any of the changes. This is the reason for the *snapshot()* and *revert()* methods.

9.18.6.6.8 snapshot() and revert() Methods

The *snapshot()* method overrides the placeholder method in the base class. It is called when the dialog object is created. It uses the *get_snapshot()* method of the Configuration object and also captures all of the starting values of the relevant properties in the *snapshot_values* list.

The *revert()* method overrides the placeholder method in the base class. It is called by the base class's *cancel()* callback method when the Cancel button is pressed. It is the opposite of the *snapshot()* method, restoring both the configuration and the properties from their snapshotted values.

9.18.6.7 Results Wizard Dialog

The Results Wizard is the GUI's most complex dialog. It is not derived from the Dialog base class, but like the Dialog class, the ResultsWizard class is derived from the tkinter Toplevel widget class.

9.18.6.7.1 Modeless Behavior

The Results Wizard dialog is not a modal window. This means that access to the main window is not blocked when it is active. This enables some very useful functionality. For example:

- It is possible to open the Preferences dialog while viewing an IV curve with the Results Wizard, allowing the user to modify the plotting preferences and immediately see the results of the changes (and to be able to either save or revert those changes)
- The View Log File and View Config File commands on the File menu are accessible, making it easy to view the log file and config file for the displayed run

Despite the fact that opening the Results Wizard does not block access to the main window, there are some specific actions that are disallowed since allowing those actions while the Results Wizard is open

is not useful, or at least the expected behavior if they were allowed is not obvious[17]. In particular, the Go button is disabled and the Results Wizard button is disabled by the *constrain_master()* method. This prevents swinging a new IV curve before closing the Results Wizard, and it prevents multiple copies of the Result Wizard from being open at the same time. Additionally, Loop Mode is disabled because that can cause some problems.

9.18.6.7.2 Results Directory

At any given time, the Results Wizard operates on a single directory that contains the results from previous runs. The path to this directory is kept in the *results_dir* instance attribute of the ResultsWizard class. The default is the application data directory. However, the value of *results_dir* can be changed to point to somewhere else that contains results from previous runs, such as a USB drive or another archive of old runs (even *really* old, including runs created by IVS1).

9.18.6.7.3 Widgets

The body of the ResultsWizard dialog is a tkinter.ttk.Frame object containing three widget types:

- tkinter.ttk.Treeview
- tkinter.ttk.Scrollbar
- tkinter.ttk.Button

Additional widgets are added in when the Overlay button is pressed. Those will be described in Section 9.18.6.7.4 on page 165.

Figure 9-16: Results Wizard Dialog

9.18.6.7.3.1 Results Treeview and Scrollbar

The *treeview()* method creates the tree view object and its scrollbar. Next, it binds the object's <<TreeviewSelect>> virtual event to the *select()* method and configures its height and width. It then calls the *populate_tree()* method, which lists the subdirectories of the results directory sorted in reverse order. This puts the run directories in newest-to-oldest order because their names are date/time strings.

[17] Strictly speaking, this means the Results Wizard is "semi-modal", if that's a thing.

For each subdirectory, it then calls either the *populate_runs()* method, the *populate_overlays()* method, or ignores the subdirectory (e.g. the "logs" subdirectory).

The *populate_runs()* method is called for each directory named "yymmdd_hh_mm_ss" in the results directory; these are the run directories. Unless it already exists, it adds a top level item to the tree view called "mm/dd/yy" (e.g. 08/20/19) using the *insert()* method of the tkinter.ttk.Treeview widget. This is the date of the run. The run is then added as a child item of the date item, using *insert()*, with the textual label of "hh:mm:ss" (e.g. 13:56:38). If the results directory contains an IV_Swinger2.cfg file that specifies a value for the title that is not "None", then the title is included after the time in the textual label.

The *populate_overlays()* method is called only for the subdirectory named "overlays" (if it exists). It adds a top level item to the tree view called "Overlays" using the *insert()* method of the tkinter.ttk.Treeview widget. It then lists its subdirectories sorted in reverse order, which puts them in newest-to-oldest order because the overlay names are date/time strings. For each subdirectory, it uses *insert()* to add it as a child of the "Overlays" item. Each item's textual label is "Created on mm/dd/yy at hh:mm:ss".

After calling the *populate_runs()* and *populate_overlays()* methods, the *populate_tree()* method calls the *heading()* method of the tkinter.ttk.Treeview widget to add the path to the results directory at the top of the tree view. This *heading()* method call also registers a callback to the *change_folder()* method, allowing the user to change to a different results directory by clicking on the heading.

9.18.6.7.3.2 Buttons

The *buttons()* method creates all of the Results Wizard button objects. Like other buttons in the application, a method callback is registered via the *command* option in each object's instantiation:

- Expand All: *expand_all()*
- Collapse All: *collapse_all()*
- Change Title: *change_title()*
- Overlay: *overlay_runs()*
- View PDF: *view_pdf()*
- Update: *update_selected()*
- Delete: *delete_selected()*
- Copy: *copy_selected()*
- Make desktop shortcut: *make_shortcut()*
- Done: *done()*

Unless hyperlinked in the list above, details are documented in the comments and code.

All buttons have an associated tooltip.

9.18.6.7.4 Select Event Actions

The *select()* method is called whenever the user changes what is selected in the tree view. The most common case is that a single run is selected, in which case that run's IV curve is displayed in the main window's image pane. This is done by the *non_overlay_select_actions()* method, which usually just calls the parent IV_Swinger2_gui object's *display_img()* method to display the existing GIF file in the

run directory. However, if that file does not exist, it can generate a new one from the data points CSV file. This makes it possible to view IVS1 results using the Results Wizard. It also makes it possible to view IVS2 results that were captured in loop mode where Save Results was checked, but the "CSV only" option was chosen.

If a single existing overlay is selected, the *overlay_select_actions()* method is called which calls the parent IV_Swinger2_gui object's *display_img()* method to display the GIF file in the selected overlay directory. There is no ability to regenerate overlays.

If multiple runs or overlays are selected, the oldest one is displayed.

See the section on overlay mode below for the actions that are taken when the selection is changed when in that mode.

9.18.6.7.5 Changing Results Directory

The *change_folder()* callback method is invoked when the user clicks on the column heading containing the path name of the results directory. This method uses a tkinter.tkfiledialog.askdirectory dialog where the user can specify the path to the new results directory by navigating to it.

If there are no overlays or runs in the specified folder, but there is a subfolder named IV_Swinger2 or the parent directory is named IV_Swinger2, then *change_folder()* assumes that the user intended to select the subfolder or parent folder respectively.

When the directory has been chosen, the *populate_tree()* method is called to create the tree view for the new results directory.

The *change_folder()* method also changes the "Make desktop shortcut" button to an "Import" button. The *import_results()* callback method registered by this button causes all of the selected overlays and runs in the new results directory to be imported (copied) to the application data directory. If nothing is selected, then everything is imported. This is a convenience feature that is useful for sharing results between different computers.

9.18.6.7.6 Changing Plot Title

The "Change Title" button invokes the *change_title()* callback method. If several error checks pass, this method uses a MyTkSimpleDialog askstring dialog to get the new title from the user. It updates the tree view item with the new title, and it changes the *plot_title* property of the GUI's IV_Swinger2 object. It then calls the *redisplay_img()* method which uses the lower level modules' plotting support to recreate and display the whole image with the new title. The *redisplay_img()* method also changes the title in the configuration and saves the configuration, so the new title is "remembered".

The "Change Title" button can also be used in Overlay Mode to change the title of the overlay.

9.18.6.7.7 Overlay Mode

The "Overlay" button invokes the *overlay_runs()* callback method. If no run is selected or if more than eight runs are selected, an error message dialog is displayed and the method returns an error code. Otherwise, overlay mode is entered.

9.18.6.7.7.1 Added Widgets

Entering overlay mode adds widgets to the Results Wizard dialog via a call to the *add_overlay_widgets()* method. Figure 9-17 below highlights the added widgets.

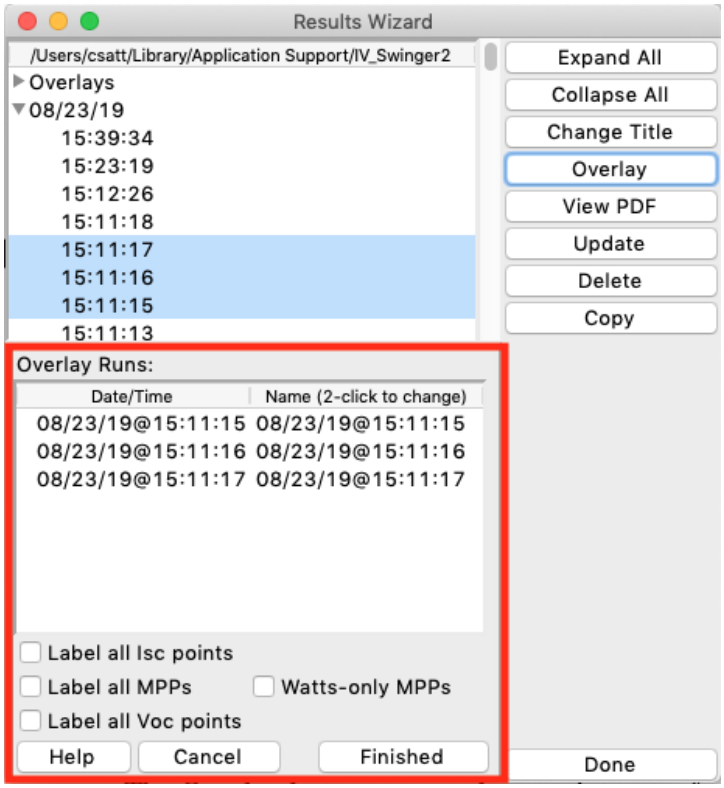


Figure 9-17: Overlay Mode Widgets

The added widgets are of the following types:

- tkinter.ttk.Label
- tkinter.ttk.Treeview
- tkinter.ttk.Checkbutton
- tkinter.ttk.Button

The *create_overlay_treeview()* method adds the tkinter.ttk.Treeview widget. Unlike the main tree view of the Results Wizard above it, this tree view has two columns and the items are not hierarchical. Each item is a single run (identified by its date and time) and its name. It is configured to allow only one item to be selected at a time (selectmode="browse"). There is no scrollbar because its height is 8, which fits the maximum number of overlays. The "Date/Time" column heading is configured to call the *chron_sort_overlays()* method when it is clicked. That method sorts the overlays chronologically, reversing the order each time it is called. The "Name" column heading is configured to call the *overlay_tv_col1_help()* method, which displays a simple help dialog on how to rename and reorder the items. The <ButtonPress-1>, <B1-Motion> and <ButtonRelease-1> events are bound to the *grab_overlay_curve()*, *move_overlay_curve()* and *update_overlay_order()* callback methods respectively to support drag-and-drop reordering. The <Double-ButtonPress-1> event is bound to the *change_overlay_curve_name()* method to support renaming the curves.

The *add_overlay_label_cbs()* method adds the four label control tkinter.ttk.Checkbutton widgets. All of these use the same callback method: *overlay_label_changed_actions()*. Each checkbutton object has an associated tooltip.

The three tkinter.ttk.Button objects are created directly in the *add_overlay_widgets()* method. A method callback is registered via the *command* option in each object's instantiation: *overlay_help()*, *overlay_cancel()* and *overlay_finished()*.

9.18.6.7.7.2 Overlay Directory

After adding the overlay widgets, the *overlay_runs()* method calls the *make_overlay_dir()* method to create the directory for the overlay image files. This is a "yymmdd_hh_mm_ss" directory under the "overlays" directory in the results directory. The directory is named for current time when the method is called, not the time of any of the overlaid runs.

9.18.6.7.7.3 Populating the Tree View

The *overlay_runs()* method calls the *populate_overlay_treeview()* method to add an item to the overlay tree view for each run that is selected in the main Results Wizard tree view.

9.18.6.7.7.4 Creating and Displaying the Overlay

After all of the above, the *overlay_runs()* method creates the overlay and displays it in the main window's image pane. This consists of first calling the *get_selected_csv_files()* method to populate the *selected_csv_files* list (instance attribute of ResultsWizard), then calling the *plot_overlay_and_display()* and *add_new_overlay_to_tree()* methods.

The *plot_overlay_and_display()* method calls the *plot_overlay()* method which creates an IV_Swinger2_plotter object and uses it to generate the overlay image (GIF). It then calls the *display_img()* method of the Results Wizard's parent GraphicalUserInterface object, which displays it in the main window's image pane.

The *add_new_overlay_to_tree()* method adds the newly created overlay to the main Results Wizard tree view under the Overlays top-level tree item.

9.18.6.7.7.5 Modifying the Overlay

When overlay mode is entered, the *overlay_runs()* method creates an overlay as described in the previous sections. While still in overlay mode, the user may do several things to modify the overlay:

- Add runs to the overlay
- Remove runs from the overlay
- Reorder the runs in the overlay
- Change the name of a run in the overlay
- Change the labeling options
- Change the title of the overlay

Adding and removing runs from the overlay is done by changing the selection in the main Results Wizard tree view. The *select()* method calls the *overlay_runs()* method when overlay mode is active; the

user does not have to click the Overlay button again. This is the same *overlay_runs()* method described in the previous sections. However, when it is invoked when overlay mode is already active, it does not call the *add_overlay_widgets()* or *make_overlay_dir()* methods, but it does repopulate the overlay tree view, creates the overlay, and displays the image.

Reordering the runs in the overlay is done by clicking on the first column header of the overlay tree view or by dragging and dropping its items. In the first case, the *chron_sort_overlays()* method is called. In the latter case, the *update_overlay_order()* method is called. Both call the *reorder_selected_csv_files()* and *plot_overlay_and_display()* methods.

Changing the name of a run in the overlay is done by double-clicking on its item in the tree view. This invokes the *change_overlay_curve_name()* method which uses a tkSimpleDialog askstring dialog to get the new name from the user. It then calls the *populate_overlay_treeview()* and the *plot_overlay_and_display()* methods.

The four checkbuttons are used to change the defaults for how the I_{SC}, MPP, and V_{OC} points are labeled. The *overlay_label_changed_actions()* callback method is invoked when any of these checkbuttons is changed. This method simply calls the *plot_overlay_and_display()* method. The current state of each checkbutton is always queried in the *plot_overlay()* method, so this has the desired effect, and the image is regenerated with the new labeling option.

Changing the title of the overlay is done with the "Change Title" button on the main Results Wizard. This calls the *change_title()* method, which uses a MyTkSimpleDialog "askstring" dialog to get the new title from the user. It then calls the *plot_overlay_and_display()* method.

9.18.6.7.7.6 Canceling the Overlay

When the Cancel button is clicked, the *overlay_cancel()* callback method is invoked. This method exits overlay mode and calls the *remove_overlay_widgets()* method. It also removes the overlay child item from the "Overlays" item in the main Results Wizard tree view and removes the overlay directory.

9.18.6.7.7.7 Finishing the Overlay

When the Finish button is clicked, the *overlay_finished()* callback method is invoked. This method first calls the ResultsWizard *plot_graphs_to_pdf()* method, which is a wrapper around the IV_Swinger2_plotter method of the same name. The wrapper adds the option for the user to retry if the file is already open in a viewer (which causes an error on Windows systems).

After generating (or attempting to generate) the PDF, the *overlay_finished()* method exits overlay mode and calls the *remove_overlay_widgets()* method. Then it calls the *rm_overlay_if_unfinished()* method which removes the overlay directory if it doesn't contain an overlaid PDF.

The *overlay_finished()* method ends by de-selecting the runs in the main Results Wizard tree view and then selecting the newly created overlay under the "Overlays" top-level item.

9.18.6.7.8 Viewing a PDF

The "View PDF" button invokes the *view_pdf()* callback method. This method uses the *sys_view_file()* function to open the PDF file corresponding to the currently displayed GIF image. In overlay mode, the

PDF is not normally generated until the Finished button is pressed, so the *plot_graphs_to_pdf()* method has to be called first. If no PDF file exists, an error dialog is displayed.

9.18.6.7.9 Batch Update

The "Update" button invokes the *update_selected()* callback method. This method displays an error dialog and returns if any overlays are selected in the tree view or if the Results Wizard is in overlay mode. Otherwise, it loops through all of the selected runs regenerating and redisplaying each image using the current plotting preferences. Only the plotting preferences are updated; the calibration values, Arduino preferences, title, etc. are preserved. See the code and comments for more details.

9.18.6.7.10 Deleting Runs

The "Delete" button invokes the *delete_selected()* callback method. This method displays an error dialog and returns if nothing is selected or if the Results Wizard is in overlay mode. Otherwise, it makes lists of all the selected overlays and runs and generates a tkinter.messagebox.askyesno dialog to get the user's permission to delete them. If the user grants permission, the send2trash function is used to move the selected overlay and/or run directories to the trash. They are then removed from the tree view.

9.18.6.7.11 Copying Runs

The "Copy" button invokes the *copy_selected()* callback method. This method displays an error dialog and returns if nothing is selected or if the Results Wizard is in overlay mode. Otherwise, it calls the *get_copy_dest()* method which uses a tkinter.tkfiledialog.askdirectory dialog where the user can specify the path to the copy destination directory by navigating to it. The *copy_overwrite_precheck()* method is then called to check if any of the directories under the destination would be overwritten and to ask if the user wants to overwrite them or not. Then all of the selected overlays and/or runs are copied to the destination directory by the *copy_dirs()* method, overwriting existing directories if the user granted permission. The *display_copy_summary()* method then generates an informational dialog summarizing the number of overlays and runs that were copied.

9.18.6.7.12 Making a Desktop Shortcut

The "Make Desktop Shortcut" button invokes the *make_shortcut()* callback method to create a shortcut on the desktop that points to the application data directory. This method first finds the path to the desktop. Then it uses code from a Stack Overflow answer to create a Windows shortcut, or just creates a symlink for Mac (or Linux). If the shortcut already exists, an error dialog is generated and the existing shortcut is left alone.

9.18.7 Exception Handling

"Anticipated" exceptions in the application code are caught with standard Python try/except handling and handled gracefully (e.g., generating an error dialog).

Once the *run()* method has invoked the *mainloop()* method of the root Tk object, the only way for application code to be executed (directly or indirectly) is a callback due to some GUI event. Exceptions that are otherwise uncaught in such code cause the callback to prematurely terminate, but do not cause the application to crash. By default, tkinter prints a message with a stack trace to stderr and the GUI

keeps running. This is OK in a development environment where the application is being run from the command line of a terminal window and the user is watching that window. But end users run the program as an installed application, invoking it by clicking on an icon, and there is no accompanying console window. With the default handling, exceptions are completely silent. Other than possibly noticing that something they were trying to do did not work correctly, users quite likely would not notice that anything went wrong at all. And if they did, there is no information with which to debug the problem.

This problem and its solution are discussed in a Stack Overflow thread. Overriding the *report_callback_exception()* method of the root Tk object makes it possible to bring the exception to the user's attention and provide the necessary debug information. The *report_callback_exception()* method of the GraphicalUserInterface class does the following:

- Logs the exception and stack trace
- Generates an error dialog asking the user to please send the log file

The *set_root_options()* method overrides the root's *report_callback_exception()* method to call that method instead of its default version. Note that, as with the default behavior, the application does not crash.

There is also the possibility that an exception is detected before *mainloop()* is running. This is handled in the *main()* function of IV_Swinger2_gui.py by using try/except to catch any otherwise uncaught exception that occurs in the creation of the GraphicalUserInterface object, or in the execution of its *run()* method. The *handle_early_exception()* global function writes the exception messages to a temporary file and uses the system file viewer to show its contents to the user.

9.19 Simulation

The IV_Swinger2_sim.py module implements the simulator. Its primary purpose is to facilitate choosing component values for scaled versions of the IV Swinger 2 hardware. It generates a simulated IV curve based on specified values for the I_{SC} and V_{OC} as well as the values of the resistors and capacitors used in the hardware. This makes it very easy to visualize the effects of different component values on the resolution and range. It also calculates the time that it takes to swing the curve and to drain the load capacitors. Component ratings are validated against the actual voltage, current and power that they will be required to handle.

The simulated IV curve is generic from the following equation:

$$I = I_{SC} - A(e^{BV} - 1)$$

This equation does not account for series or parallel resistance, and the A and B coefficients are chosen such that the MPP current and voltage are at a typical ratio of the I_{SC} and V_{OC}, respectively. This results in a fairly representative curve that should be adequate to predict the resolution and other characteristics of real IV curves with the given I_{SC} and V_{OC}. Of course, the shape of the real IV curves will differ depending on actual series and parallel resistances, temperature, etc.

The simulator also supports the automated choice of optimal components for a given maximum Isc and Voc. The user can then test these component values against smaller I_{SC} and V_{OC} values.

This module may be used standalone, or it may be imported.

Most of this module consists of the following classes:

- IV_Swinger2_sim
- SimulatorDialog
- SimulatorHelpDialog

The IV_Swinger2_sim class has properties for all of the configurable inputs (I_{SC}, V_{OC}, component values, etc). Its *simulate()* method uses these to generate the simulated IV curve in the form of ADC values, i.e. the actual "points" that the hardware would measure. Its *run()* method calls the *simulate()* method and then plots its IV curve. Like a real run, the results are saved in a run directory. This includes a configuration file. This makes it possible to view the results later with the Results Wizard.

The SimulatorDialog class is the tkinter/tkinter.ttk GUI interface to the IV_Swinger2_sim class. It is designed so that it may be run as the child of any tkinter.ttk.Frame object (namely a GraphicalUserInterface object from IV_Swinger2_gui.py, but not necessarily).

There is also a *main()* function that is used when the module is run standalone. The *main()* function creates a SimulatorDialog object with a basic tkinter.ttk.Frame object as its parent and runs it.

The IV_Swinger2_sim.py module is over 3k lines long (2k physical lines of code). Ideally, this section of the document would describe its implementation in much more detail than the above, but due to time constraints, the comments and docstrings in the Python code will have to suffice for now.

9.20 PV Modeling

Generic PV modeling is implemented in the IV_Swinger_PV_model.py module. The IV_Swinger2_PV_model.py module adds features specific to IV Swinger 2, and that is used by the IV_Swinger2.py and IV_Swinger2_gui.py modules.

9.20.1 IV_Swinger_PV_model.py

The IV_Swinger_PV_model.py module implements modeling of PV cells and modules. It is part of the IV Swinger project, but has no dependencies on other code from the project. Therefore, it may be imported and used for other unrelated projects without importing the other IV Swinger modules.

The main purpose of this module is to predict the IV curve of a PV module/cell given its datasheet values and the irradiance and cell temperature conditions under which it is operating. When compared to a measured IV curve, this "reference" IV curve can be used to evaluate the performance of the PV under test. This modeling is not a trivial task. There are many research papers devoted to the topic. I primarily studied the following papers:

Ibrahim, Haider & Anani, Nader. (2017)
"Variations of PV module parameters with irradiance and temperature."
Energy Procedia. 134. 10.1016/j.egypro.2017.09.617

De Soto, W. & Klein, S.A. & Beckman, W.A. (2006)
"Improvement and validation of a model for photovoltaic array performance."
Solar Energy. 80. 78-88. 10.1016/j.solener.2005.06.010.

The code (like most of the research papers) uses the "single-diode" circuit model for PV cells. A 2-diode circuit model is slightly more accurate, but makes the mathematics too complex to be justified (and it is difficult enough with the single-diode model). The following equation defines the relationship between current and voltage for the single-diode model.

$$I = I_L - I_0 \cdot \left[e^{\frac{V + I \cdot R_S}{n \cdot N_S \cdot V_{TH}}} - 1 \right] - \frac{V + I \cdot R_S}{R_{SH}}$$

where:

$I = output\ current$
$V = output\ voltage$
$I_L = light\ current, aka\ photocurrent$
$I_0 = diode\ reverse\ saturation\ current$
$R_S = series\ resistance$
$R_{SH} = shunt\ (parallel)\ resistance$
$n = diode\ ideality\ factor$
$N_S =\ number\ of\ series\ connected\ cells$
$V_{TH} = thermal\ equivalent\ voltage = \frac{kT}{q}$

Where:

$k = Boltzmann\ constant\ \left(1.391\ \cdot\ 10^{-23}\frac{J}{K}\right)$
$T = cell\ temperature\ (K)$
$q = charge\ of\ an\ electron\ (1.602\ \cdot\ 10^{-19}C)$

$\therefore V_{TH} =\ 25.7\ mV\ at\ 25\ ^\circ C\ (298.15\ K)$

At a given cell temperature, the value ($n*N_S*V_{TH}$) is constant, which we will name "A":

Equation 9-1: Single-diode model equation

$$I = I_L - I_0\ \cdot \left[e^{\frac{V + I \cdot R_S}{A}} - 1 \right] - \frac{V + I \cdot R_S}{R_{SH}}$$

What makes this equation difficult to work with is the fact that the current (I) is on both sides of the equation and it is in the exponent of *e* on the right side. This makes it both "implicit" and "transcendental". Algebra cannot be used to solve for I, given the other values. Instead, numerical methods (such as the Newton-Raphson method) must be used to iteratively search for the solution[18]. Fortunately, the SciPy library provides the tool we need - namely a "root solver".

The values of the following five parameters need to be determined in order to generate an IV curve:

- I_L, I_0, A, R_S and R_{SH}

[18] This isn't quite true. It is possible to use the Lambert W function to make the equation explicit.

Their values are dependent on the particular PV module/cell characteristics and also on the irradiance and cell temperature.

The PV datasheet provides the following four values at standard test conditions (aka STC, which are 1000 W/m^2 irradiance and 25 ˚C cell temperature):

- V_{OC} = open-circuit voltage
- I_{SC} = short-circuit current
- V_{MP} = voltage at maximum power point
- I_{MP} = current at maximum power point

Knowing these values for the STC curve is enough information to determine the five parameter values for the STC curve, and knowing their values for the STC curve allows us to calculate their values for other non-STC cell temperatures and irradiances.

Determining the five parameter values for the STC curve requires simultaneously solving five equations. These five equations are based on information we know about three points on the STC curve: the V_{OC} point, the I_{SC} point, and the maximum power point (MPP).

The first equation is based on the fact that we know that I=0 and V=V_{OC} at the V_{OC} point. Substituting 0 for I and V_{OC} for V in Equation 9-1 results in the following equation:

Equation 9-2: Simultaneous equation #1 (V_{OC})

$$I_L - I_0 \cdot \left[e^{\frac{V_{OC}}{A}} - 1\right] - \frac{V_{OC}}{R_{SH}} = 0$$

The second equation is based on the fact that we know that V=0 and I=I_{SC} at the I_{SC} point. Substituting 0 for V and I_{SC} for I in Equation 9-1 results in the following equation:

Equation 9-3: Simultaneous equation #2 (I_{SC})

$$I_L - I_0 \cdot \left[e^{\frac{I_{SC} \cdot R_S}{A}} - 1\right] - \frac{I_{SC} \cdot R_S}{R_{SH}} - I_{SC} = 0$$

The third equation is based on the fact that we know that I=I_{MP} and V=V_{MP} at the MPP. Substituting I_{MP} for I and V_{MP} for V in Equation 9-1 results in the following equation:

Equation 9-4: Simultaneous equation #3 (MPP)

$$I_L - I_0 \cdot \left[e^{\frac{V_{MP} + I_{MP} \cdot R_S}{A}} - 1\right] - \frac{V_{MP} + I_{MP} \cdot R_S}{R_{SH}} - I_{MP} = 0$$

Now things get more complicated. The fourth equation is based on the fact that we know that the MPP is in fact the point with the maximum power. The power curve (power vs voltage) peaks at the MPP, so its slope is zero at that point. This can be expressed as: *dP/dV*=0 at the MPP. Since P=IV, we need *d(IV)/dV* to be zero when I=I_{MP} and V=V_{MP}. Using the product rule of differentiation:

$$\frac{d(IV)}{dV} = I \cdot \frac{dV}{dV} + V \cdot \frac{dI}{dV} = \mathrm{I} + V \cdot \frac{dI}{dV} = 0 \qquad @\ \mathrm{V}=V_{MP},\ \mathrm{I}=I_{MP}$$

Now we need to find *dI/dV* by differentiating Equation 9-1. Since it is an implicit function, this requires using implicit differentiation. Fortunately, there's a great tool to do this: https://www.derivative-calculator.net/. The "Options" tab has a checkbox for implicit differentiation, which must be checked. Using "x" instead of "V", "y" instead of "I", and k_il, k_io, k_a, k_rs and k_rsh for the five parameters, Equation 9-1 is expressed to the tool as:

```
y = k_il-k_io*(e^((x+y*k_rs)/k_a)-1)-(x+y*k_rs)/k_rsh
```

That generates this output when the Go! button is pressed:

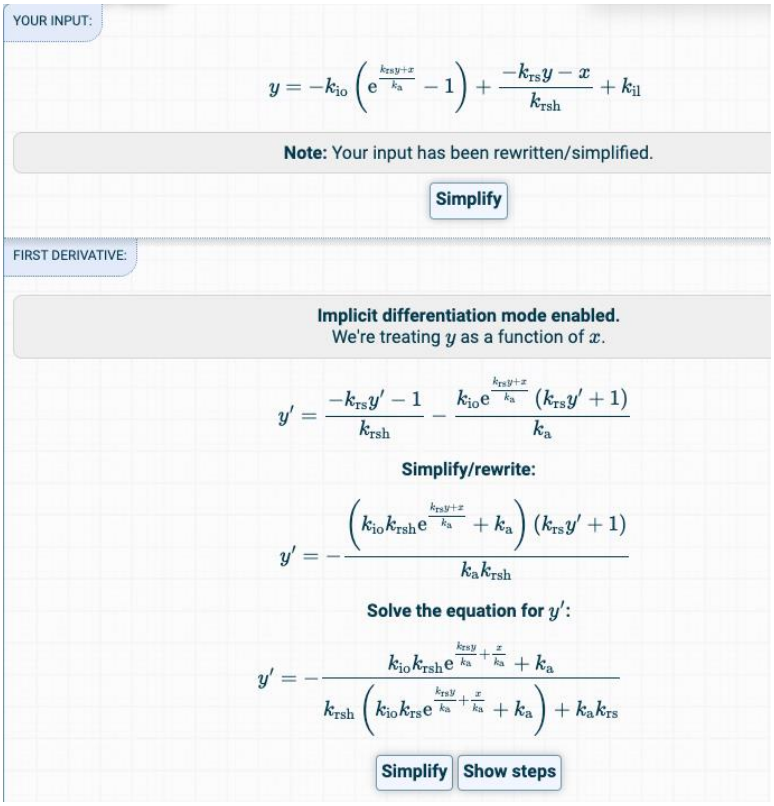


Figure 9-18: *dI/dV* (x=V, y=I) using derivative-calculator.net

Notice the minus sign! The "Show steps" button can be used to display the detail, including tooltips that show the rules applied. Very nice.

Now we just need to convert the variables and constants back to our names and substitute I_{MP} for I and V_{MP} for V, and we finally have our fourth equation:

Equation 9-5: Simultaneous equation #4 (*dP/dV*=0 @ MPP)

$$I_{MP} - V_{MP} \cdot \frac{I_0 \cdot R_{SH} \cdot e^{\frac{V_{MP}+I_{MP}\cdot R_S}{A}} + A}{R_{SH} \cdot \left[I_0 \cdot R_S \cdot e^{\frac{V_{MP}+I_{MP}\cdot R_S}{A}} + A\right] + R_S \cdot A} = 0$$

The fifth equation is based on the fact (or at least approximation) that the slope at the I_{SC} point is the negative reciprocal of R_{SH}. An infinite R_{SH} produces a curve that starts out completely horizontal, and a finite R_{SH} results in a curve that slopes down from the I_{SC} point. This can be expressed as:

$$\frac{dI}{dV} = -\frac{1}{R_{SH}} \quad @\, V = 0, I = I_{SC}$$

For equation #4 we used the derivative-calculator.net tool to find *dI/dV* by differentiating Equation 9-1, so we can re-use that result from Figure 9-18 with 0 substituted for V and I_{SC} substituted for I. This gives us the fifth and final equation we need:

Equation 9-6: Simultaneous equation #5 (*dI/dV* = -1/R_{SH} @Isc)

$$\frac{1}{R_{SH}} - \frac{I_0 \cdot R_{SH} \cdot e^{\frac{I_{SC} \cdot R_S}{A}} + A}{R_{SH} \cdot \left[I_0 \cdot R_S \cdot e^{\frac{I_{SC} \cdot R_S}{A}} + A \right] + R_S \cdot A} = 0$$

The SciPy root solver is used for solving these simultaneous equations to determine the values of I_L, I_0, A, R_S, and R_{SH}. However, there are cases where this fails to converge. In some such cases using only the first four equations with a fixed value for R_{SH} converges and produces a good result. And in some cases, equation #4 must be ignored altogether for the solver to converge. In that case, the result is imperfect because there is a point on the modeled curve that has a higher power than the specified MPP. But the curve does pass through the specified MPP, and the modeled curve is usable for most purposes.

To generate the IV curve at STC, the parameters are derived using the STC values from the datasheet. That is nice, but not very useful other than to validate the model since the STC IV curve is generally included in the datasheet anyway. What we really want is to generate the IV curve at non-STC values of irradiance and/or cell temperature.

The effect of irradiance is modeled as a scaling of the light current (I_L) in proportion to the STC irradiance. It assumes that irradiance does not affect I_0, A, R_S or R_{SH}. This is fairly accurate except at low irradiance.

The effect of cell temperature is determined from the following datasheet values:

I_{SC} temperature coefficient (%/˚C or mA/˚C)
V_{OC} temperature coefficient (%/˚C or mV/˚C)
MPP temperature coefficient (%/˚C or W/˚C)

Note that the MPP temperature coefficient specifies a power delta, and does not split out its current and voltage components. We assume that the current component is equal to the Isc temperature coefficient, and the voltage component is derived based on that and the power coefficient.

A two-step process is used to generate the IV curve at a given irradiance and cell temperature:

Step 1 (account for temperature only):
- Calculate temperature-adjusted I_{SC} (@ 1000 W/m^2)
- Calculate temperature-adjusted V_{OC} (@ 1000 W/m^2)
- Calculate temperature-adjusted V_{MP} (@ 1000 W/m^2)

- Calculate temperature-adjusted I_{MP} (@ 1000 W/m^2)
- Use root solver to determine I_L, I_0, A, R_S and R_{SH}

Step 2 (adjust for irradiance):
- scale I_L from step 1 by irradiance/1000 W/m^2
- Use I_0, A, R_S and R_{SH} from step 1
- Generate curve using root solver

This module also supports a reverse computation, where the measured V_{OC} and I_{SC} are provided and the temperature and/or irradiance are derived.

9.20.1.1 SciPy root solver usage

The SciPy root solver finds the roots of a user-provided "vector function":

scipy.optimize.**root**(*fun, x0, args=(), method='hybr', jac=None, tol=None, callback=None, options=None*)

The vector function, *fun*, is a function that is called with an array of N values and returns an array of N values. The root solver's goal is to find the N input values that produce zeroes on all of the N output values. To find the root of a single equation, a function that takes one input value and produces one output value is passed to the root solver. To find the roots of N simultaneous equations, a function that takes N input values and produces N output values (using the N equations) is passed to the root solver.

The *x0* parameter is an array of N initial guesses for the roots. The root solver's success depends heavily on being provided with "good" initial guesses. Surprisingly, guesses that are closest to the final solution values are not always the best. The PV modeling code uses ordered lists of initial guesses for each parameter, and tries different combinations in nested loops until the root solver succeeds. The order of the initial guess lists was determined empirically, with the most commonly successful guesses (for the set of sample PVs) coming first, and others in decreasing order of their success.

The *args()* parameter are "extra" arguments that are passed to *fun*. These arguments are constants for a given call to **root**().

The defaults for the remaining parameters are used by the PV modeling code. Some experimentation was performed providing a Jacobian function (*jac* parameter), but it did not improve the performance or results. Determining the Jacobian function requires NxN partial derivatives, so it is also mistake-prone.

9.20.1.2 Global functions

The global functions named *test_** are all constructed to be compatible with scipy.optimize.**root** as its *fun* argument. However, only the following are actually used as such directly:

- *test_i_given_v_and_parms(amps, volts, il_i0_a_rs_rsh)*
- *test_voc(voc, il_i0_a_rsh)*
- *test_isc(isc, il_i0_a_rs_rsh)*
- *test_mpp(vmp_imp, il_i0_a_rs_rsh)*
- *test_parms(il_i0_a_rs_rsh, voc_isc_vmp_imp_ignore_eq4)*
- *test_first_four_parms(il_i0_a_rs, rsh_voc_isc_vmp_imp_ignore_eq4)*

In all cases, the first parameter contains the value(s) being solved for by the root solver, and the remaining parameter(s) contain the constant values passed to the root solver in its *args()* parameter. For example, *test_mpp* is passed a list of fixed values for I_L, I_0, A, R_S and R_{SH} in the second argument. The root solver then tries different values for V_{MP} and I_{MP} in the first argument until the value returned by *test_mpp* is close to [0, 0]. Internally, *test_mpp* applies all of the input values to equation #3 (Equation 9-4) and equation #4 (Equation 9-5) to generate the two output values.

The *test_eq3*, *test_eq4* and *test_eq5* functions are used internally by *test_parms* and *test_first_four_parms*.

The *find_parms* function uses the SciPy root solver to find the values of the I_L, I_0, A, R_S and R_{SH} parameters (N=5).

The *find_parms* function takes the following as inputs:

- V_{OC}, I_{SC}, V_{MP}, I_{MP}
- A single guess for I_L (usually equal to I_{SC})
- A list of guesses for I_0
- A single guess for A
- A list of guesses for R_S
- A list of guesses for R_{SH}

It then loops, calling the root solver with the different combinations of guesses. Since this can be time-consuming, it declares success and terminates if a solution is found that is "good enough". It is "good enough" if none of the equations has an absolute value greater than the *err_thresh* parameter provided by the caller. Performance is optimized if the guesses are ordered from most to least likely to succeed.

The outermost loop first tries all the inner loops with *ignore_eq4* set to False. The second-to-outermost loop first tries all the inner loops with *use_eq5* set to True. Ideally, a solution is found before either of these loops repeats, meaning all five equations are satisfied within a margin of at least *err_thresh*. If not, then the second-to-outermost loop sets *use_eq5* to False, which causes the innermost loop to run the root solver only for the first four equations and does not solve for R_{SH}, using the current R_{SH} guess as a given. If that too fails, then the outermost loop sets *ignore_eq4* to True, which causes the root solver to be "fooled" into thinking that equation #4 is always satisfied. This results in an imperfect modeling, but usually better than nothing.

The remaining global functions: *pv_spec_from_dict*, *read_pv_specs*, *add_pv_spec*, *check_pv_spec* and *create_pv_spec_file* are support functions for creating and using a “PV spec CSV file” that contains the names and relevant specifications for a collection of PV modules and/or cells.

9.20.1.3 PV_model class

The only class in the IV_Swinger_PV_model.py module is the PV_model class. It has properties that the user populates with the irradiance, cell temperature and values from the PV spec. Then the *run()* method is called to determine the single-diode model parameters. Other methods are provided to generate points on the curve and to provide other utility tasks.

9.20.1.3.1 Properties

The PV_model class has the following input properties that the user must populate based on the PV spec before running the model:

- *pv_name*
- *voc_stc*
- *isc_stc*
- *vmp_stc*
- *imp_stc*
- *num_cells* (optional, estimated using *voc_stc* if not provided)
- *voc_temp_coeff_pct_per_deg*
- *voc_temp_coeff_pct_per_deg*
- *mpp_temp_coeff_pct_per_deg*

Before running the model, the following input properties must be also populated by the user:

- *irradiance*
- *cell_temp_c*

The following input properties have default values that are usually adequate, but may be overridden by the user:

- *i0_guesses*
- *rs_guesses*
- *rsh_guesses*
- *err_thresh*

The following are output properties that are valid only after the model has been run:

- *il*
- *i0*
- *a*
- *rs*
- *rsh*
- *vmp*
- *imp*

The PV_model class also has a *vi_points[]* output property, which is a list of points on the modeled curve. This is not populated until the *add_vi_points()* method is run.

The *run_ms* output property captures the number of milliseconds that the *run()* method took to execute.

9.20.1.3.1.1 Derived properties

The *voc_temp_coeff_mv_per_deg* and *isc_temp_coeff_ma_per_deg* derived properties can be used to set the *voc_temp_coeff_pct_per_deg* and *isc_temp_coeff_pct_per_deg* properties using the alternate units specified in some datasheets.

The *a_guess* derived property uses the number of cells and cell temperature to estimate the value of the A parameter, assuming the ideality factor (n) is IDEALITY_FACTOR_GUESS (currently 1.0).

The *cell_temp_k, temp_diff_from_stc, isc_at_temp, voc_at_temp, imp_at_temp* and *vmp_at_temp* derived properties perform simple calculations based on the *cell_temp_c, isc_stc, voc_stc, imp_stc, vmp_stc, isc_temp_coeff_pct_per_deg, voc_temp_coeff_pct_per_deg, isc_temp_coeff_pct_per_deg* and *mpp_temp_coeff_pct_per_deg* properties.

The *voc* and *isc* derived properties use the root solver with the *test_voc* and *test_isc* functions respectively to determine V_{OC} and I_{SC}, based on the *il, i0, a, rs,* and *rsh* properties.

The *ideality_factor* derived property calculates the ideality factor (n) based on the *a, num_cells, and cell_temp_k* properties. If *num_cells* is not set, it is estimated based on the *voc_stc* property.

The *parms_string, parms_string_w_newlines, title_string,* and *summary_string* derived properties are utilities for logging and other display of the model inputs and results.

9.20.1.3.2 Methods

9.20.1.3.2.1 run()

The most important PV_model class method is the *run()* method, which runs the model once the input properties have been populated (see section 9.20.1.3.1 above). Once this method has been executed, the output properties with the single-diode model parameters will contain their derived values and the *voc, isc, vmp,* and *imp* output properties will also return the correct values.

If the modeling fails to find a solution, an *AssertionError* exception is raised.

If the "solution" required ignoring Equation #4 (by the *find_parms* function), no exception is raised, but a True value is returned by the method.

9.20.1.3.2.2 gen_vi_points(), add_vi_points() and print_vi_points()

The *gen_vi_points()* method is a Python generator. It returns an iterator yielding V,I points for the modeled curve. This generator can be used only after a successful execution of the *run()* method. Each point is yielded as a (v,i) tuple. Its only parameter is the number of points to generate. For each voltage value, it uses the root solver and the *test_i_given_v_and_parms()* global function (which implements Equation 9-1) to determine the corresponding current value. The voltage values are not evenly distributed between 0 and V_{OC} volts. Instead, voltage increments are proportional to the square root of the point number. This results in large voltage increments at the I_{SC} end of the curve and very small voltage increments at the V_{OC} end. This gives better resolution around the MPP and also on the steep tail end of the curve where small voltage increments map to large current increments. Since that probably won't include the actual MPP, the MPP is yielded before yielding the first point with a voltage higher than V_{MP}.

The *add_vi_points()* method uses the *gen_vi_points()* method to populate the *vi_points[]* property. The *print_vi_points()* method calls the *add_vi_points()* method if the *vi_points[]* property is not already populated, and then prints each of the points. Note that neither *add_vi_points()* nor *print_vi_points()* is used by the higher-level IV Swinger 2 modules.

9.20.1.3.2.3 estimate_irrad(), estimate_temp_from_irrad(), estimate_temp() and estimate_irrad_and_temp()

The *estimate_irrad()* method estimates irradiance, given a measured I_{SC} value. The *irradiance* property is updated with the estimate. This method requires the *cell_temp_c* property to be valid (or at least a valid guess). The calculation is based on the *isc_stc* and *isc_temp_coeff_pct_per_deg* properties, which along with the measured I_{SC} and cell temperature are enough to infer the irradiance.

The *estimate_temp_from_irrad()* method estimates cell temperature from the *irradiance* property, given a measured I_{SC}. The calculation is based on the *isc_stc* and *isc_temp_coeff_pct_per_deg* properties, which along with the measured I_{SC} and irradiance are enough to infer the cell temperature.

The *estimate_temp()* method estimates cell temperature, given both measured V_{OC} and I_{SC} values and a valid *irradiance* property (or at least a valid guess). First, the *estimate_temp_from_irrad()* method is called to estimate the cell temperature from the measured I_{SC} and the *irradiance* property. Then the *run()* method is called with the cell temperature and irradiance estimates. Finally, the temperature error is calculated based on the measured and modeled V_{OC} and the *cell_temp_c* property is adjusted accordingly.

The *estimate_irrad_and_temp()* method estimates both irradiance and cell temperature, given measured values for V_{OC} and I_{SC}. This uses an iterative algorithm. The first step for each iteration is to estimate the irradiance using the *estimate_irrad()* method. This is based on the estimated temperature and the measured I_{SC}. Initially, the estimated temperature is 45 ˚C, which is a typical NOCT. The temperature estimate is then updated by running the *estimate_temp()* method. The error between the previous and current estimated temperature is then calculated. The iterations continue while the error in the estimated temperature is greater than the specified threshold.

9.20.1.3.2.4 get_spec_vals() and apply_pv_spec_dict()

The *get_spec_vals()* method gets the spec values for a given PV from a CSV file by calling the *read_pv_specs()* function and updates the associated PV_model object properties by calling the *apply_pv_spec()* method.

9.20.1.3.2.5 update_mpp()

The *update_mpp()* method requires the *il*, *i0*, *a*, *rs* and *rsh* properties and the *irradiance* property to be valid. It calls the root solver with the *test_mpp()* global function to determine V_{MP} and I_{MP}, which it assigns to the *vmp* and *imp* properties. It is called by the *run()* method after the parameters have been determined.

9.20.1.4 IV_Swinger_PV_model.py *main()* Function

The IV_Swinger_PV_model.py module has a *main()* function at the end of the file. It is not used unless the module is run standalone, rather than being imported. Its purpose is more to be an example than to really be useful.

The *main()* function creates a PV_model object and then populates its input properties with the spec values for the SunPower X21-345 PV module. It also sets the *irradiance* and *cell_temp_c* properties to 800 W/m^2 and the NOCT from the datasheet. It then calls the *run()* method and the *print_vi_points()* method. The resulting V_{OC}, I_{SC} and MPP can be compared with the datasheet values for 800 W/m^2 and NOCT, validating the model (about 0.2% error in this case). Finally, it calls the

estimate_irrad_and_temp() method with the datasheet V_{OC} and I_{SC} values for 800 W/m^2 and NOCT. This demonstrates that the estimated irradiance and cell temperature are close to 800 W/m^2 and NOCT, demonstrating the ability of the model to perform the estimation of the irradiance and cell temperature based on a given V_{OC} and I_{SC}.

9.20.2 IV Swinger 2 PV Modeling

The generic IV_Swinger_PV_model.py module is extended for IV Swinger 2 by the IV_Swinger2_PV_model.py module and that module is used by the IV_Swinger2.py and IV_Swinger2_gui.py modules.

9.20.2.1 IV_Swinger2_PV_model.py

The IV_Swinger2_PV_model.py module is very small. It defines the IV_Swinger2_PV_model class, which extends the PV_model class by adding a *data_points[]* attribute, a *csv_filename* property and *get_data_points()* and *gen_data_points_csv()* methods. This is for compatibility with the IV_Swinger and IV_Swinger2 code, which require a *data_points[]* list that consists of (amps, volts, ohms, watts) tuples.

The *get_data_points()* method uses the parent class's *gen_vi_points()* generator method to generate (volts, amps) tuples, each of which it converts to (amps, volts, ohms, watts), which are written to the *data_points[]* attribute.

The *gen_data_points_csv()* method uses the IV_Swinger class's *write_csv_data_points_to_file()* method to write the data points to the file named in the *csv_filename* property.

The module has a *main()* function to allow it to be run standalone, for testing and for an example. The *main()* function calls the *create_pv_spec_file()* function, which creates a PV spec CSV file with example PV specs in the current directory. It then creates a IV_Swinger2_PV_model object and populates it with the SunPower X21-345 module's spec values by calling the inherited *get_spec_vals()* method. Next, the irradiance property is set to 800 W/m^2 and the cell temperature property is set to the module's NOCT. The *run()* and *get_data_points()* methods are then called to run the model and generate the curve with 100 points. Then the *gen_data_points_csv()* method is called. Finally, a local function *plot_and_view_modeled_curve()* is called. This function creates an IV_Swinger2_plotter object and runs it to generate the PDF. It then uses the system viewer to open the PDF.

9.20.2.2 Usage in IV_Swinger2.py

The IV_Swinger2 class instantiates an IV_Swinger2_PV_model object named *pv_model* at initialization. None of *pv_model*'s input properties are set at this time, however.

Higher level code is responsible for providing a PV spec CSV file and setting the *pv_name* property to the name of a PV whose specifications exist in the PV spec CSV file. This may be done directly or by applying the "**pv name**" config option in the [PV Model] section of the configuration file.

The *gen_reference_curve()* method uses the PV model to generate a CSV file with a reference curve corresponding to the current run that has been captured by IV Swinger 2 or an old run. The model requires the irradiance and cell temperature. These may either be from the sensor values for the run (from the run info file) or estimated by the model (based on the measured I_{SC} and V_{OC} from the run's

data points CSV file.) If sensor values do not exist, there is no choice but to use the model to perform the estimates. Even if a sensor value does exist, the estimate is still used if the *estimate_irrad* and/or *estimate_temp* property is True. The *pv_model*'s *get_spec_vals()* method is called to extract the values from the PV spec CSV file and apply them to the *pv_model* spec properties. Alternately, the higher-level code may set the *use_curr_pv_model_props* property, indicating that it has pre-populated the *pv_model* spec properties. The *pv_model*'s *run()*, *get_data_points()* and *gen_data_points_csv()* methods are then called to generate the CSV file with the reference curve's data points. The format of this CSV file is exactly the same as the data points CSV file from a run. The CSV file is written to the run directory, alongside the run's data points CSV file and run info file. The *irrad_estimated* and *cell_temp_estimated* properties are set to indicate whether the irradiance and/or cell temperature were estimated.

The *add_reference_curve()* method calls the *gen_reference_curve()* method to generate the reference curve CSV file in the run directory. Then both CSV files are passed to the plotter, with the reference curve coming first. The plotter's *curve_names* property is set such that the reference curve's name (used in the legend) shows the PV name and the irradiance and temperature that were used to generate it, including annotations as to whether these values were obtained from the sensors or were estimated. Before the plotter's *run()* method is called, the *plot_results()* method calls the *add_reference_curve()* method if the *pv_name* property is set and the *plot_ref* property is True. The plotter (IV_Swinger_plotter.py module) generates the plot just as if it were an overlay of two IV curves. The only difference is that the generated image files (GIF and PDF) are named like a normal non-overlaid case, and the legend is different.

The *gen_pv_test_curve()* method is used to generate and plot a standalone test curve for *pv_model*. The model must already have been run and the data points generated using the *get_data_points()* method before calling this method. It creates an output directory for the test curve, calls the *gen_data_points_csv()* method to deposit the test curve's CSV file in the created directory, and then calls the *plot_pv_test_curve()* method. The *plot_pv_test_curve()* method creates an IV_Swinger2_plotter object and calls its *run()* method to generate the plot. The plotter's *linear* property is set to False to force Catmull-Rom spline interpolation, which results in a more accurate MPP. Its *point_scale* property is set to 0.0 to display the curve without showing the individual points. The *curve_names* property is set to *pv_model*'s *parms_string_w_newlines* property and its *run_ms* property so the legend shows the values of all the modeled parameters and the model run time in milliseconds. Note that the curve_names property is not saved in the config file, so this legend is lost if a PV model test curve is regenerated later (e.g., from the GUI Results Wizard).

9.20.2.2.1 PV Model Configuration

The configuration [Plotting] section has a "**plot ref**" option that is used to save and restore the value of the *plot_ref* IV_Swinger2 property, which controls whether reference curves are added to measured IV curves.

The [PV Model] section has the following options to control the generation of reference curves:

- **pv name**: string with the name of the current PV to be used for reference curves
- **estimate irrad**: True if the irradiance should be estimated even when there is a measured value
- **estimate temp**: True if the cell temp should be estimated even when there is a measured value
- **use avg sensor temp**: True if multiple temperature sensors should all be averaged to determine cell temperature; False if only the first should be used
- **cell temp adjust**: ˚C to add to sensor temperature to get "measured" cell temperature

The associated IV_Swinger2 properties are: *pv_name, estimate_irrad, estimate_temp, use_avg_sensor_temp* and *cell_temp_adjust*.

9.20.2.3 Usage in IV_Swinger2_gui.py

The IV_Swinger2_gui.py module has the following functionality related to PV modeling:

- The "Plot Reference" checkbutton on the main screen controls whether the reference curve should be added to measured IV curves. It determines the value of the "**plot ref**" config option.

- The PV Model tab of the Preferences dialog is used to:
  - Set the current PV name ("**pv name**" config option)
  - Set other preferences that affect PV reference curves ("**estimate irrad**", "**estimate temp**", "**use avg sensor temp**", and "**cell temp adjust**" config options)
  - Create the PV spec CSV file and add, delete and modify its entries
  - Test the modeling for a PV at any irradiance and cell temperature

The PreferencesDialog class has a *pv_specs* attribute that is a list of PV specs (each of which is a *pv_spec_dict* object.) It is initially populated by calling the PreferencesDialog *initialize_pv_specs()* method which reads the specs from the PV spec CSV file. If the file doesn't exist, it is created. The *pv_specs[]* list is used by the PV Model tab to list the names of the PVs in the listbox and to display the specs for the PV that is currently selected. When the user makes changes to add, delete, or modify PV specs, the *pv_specs[]* list is changed, but the PV spec file is not changed unless and until the OK button is pressed, at which point the *pv_model_apply()* method calls the IV_Swinger_PV_model.py *add_pv_spec()* function.

If a measured IV curve and its reference curve are currently displayed in the main window (either the current run that has been captured by IV Swinger 2 or an old run selected in the Results Wizard), changes made to the selected PV's specs on the PV Model tab can cause the plot to be regenerated with the updated reference curve. Unlike changes on the Plotting tab that cause an "immediate" application, changes on the PV Model tab are only applied when the user clicks the Apply button or the OK button. The Apply button invokes the *pv_model_apply_button_actions()* method, which applies the changes without dismissing the dialog. Changes that are made to the displayed plot are reverted if the Cancel button is pressed after the Apply button is used. The *pv_model_apply()* method is called both by the *pv_model_apply_button_actions()* method and by the *apply()* method, which is called when the OK button is pressed. The difference is that its *update_pv_spec_file* parameter is set to False in the former case and set to True in the latter. This is important because, unlike the config, the PV spec file is not reverted when the Cancel button is pressed, so changing it needs to be suppressed.

The PV Model tab also supports testing the modeling for a PV at any irradiance and cell temperature. When the Test button is pressed, the *pv_test_button_actions()* method is invoked, which calls the *run_pv_model_test()* method. This calls the *apply_specs_to_pv_model()* method to update the IV_Swinger2 object's *pv_model* properties by calling its *apply_pv_spec_dict()* method with the current spec values from the Entry widgets. Next, the *run_pv_model_test()* method directly sets *pv_model*'s *irradiance* and *cell_temp_c* properties based on the specified test values. After that, it calls *pv_model*'s *run()* method. If the *run()* method fails (assertion), an error dialog is displayed with the assertion message and the function returns without generating or displaying a curve. If *run()* returns a value of True, a warning dialog displayed saying that the modeling was imperfect. This means that equation #4 had to be ignored for the model to find a solution. Following the call to *run()*, the model's *get_data_points()* method is called to generate the actual curve. Next, the IV_Swinger2 object's

gen_pv_test_curve() method is called, and *display_img()* is called to display the generated image in the main window. Note that each test run results in a new folder with the with the generated test curve, and it is available in the Results Wizard, along with the measured (and simulated) IV curves.

10 Software: Mac and Windows Installer Builds

It would not be reasonable to expect users to run the IV Swinger 2 application from the command line using Python. They would have to install the necessary support libraries and might even have to install Python itself. To make things as easy as possible for the end user, the application needs to be installable on their laptop using the standard method for their platform. On Mac, this means providing a disk image (DMG) file. On Windows, it means providing a Windows Installer (MSI) file.

Note: the scripts referenced in this section are found in the IV_Swinger GitHub repository under the build_tools directory. The README file in that directory contains the step-by-step instructions for both Mac and Windows builds.

10.1 PyInstaller

Before building a DMG or MSI file, the application has to be bundled with all of its dependencies, including Python itself and all of the external support libraries. The bundle is an executable package created with PyInstaller.

There are two types of PyInstaller bundles: one-folder and one-file. In our case, the difference is transparent to the user since the DMG / MSI installers are single files and take care of everything regardless of how many files there are in the package. The one-file bundle takes significantly longer to start up, especially on older laptops, so the one-folder option is used.

PyInstaller must be run on the target platform. Furthermore, running it on an older version of that platform assures that the generated executable will run properly on laptops running that OS or newer. I build the released Mac executables on an old MacBook running Yosemite (10.10) and build the released Windows executables on an old Windows 7 laptop. The Mac Yosemite build doesn't support Python 3 or the newer Tcl/Tk versions, which are required for dark mode support, so a second Mac executable is built on Mojave (10.14) for releases starting with 2.7.0.

10.1.1 Platform-Specific Scripts

The mac_run_pyi bash script runs PyInstaller for the Mac, and the run_pyi.bat script runs it for Windows. They both:

- Use the --windowed option
- Use the --noconfirm option
- Use the --add-data option to include the files:
  - Splash_Screen.png
  - Blank_Screen.png
  - version.txt
- Use the --name option to name the app "IV Swinger 2"
- Use IV_Swinger2_gui.py as the target program

10.1.1.1 mac_run_pyi

The mac_run_pyi script uses the --icon option to point to the IV_Swinger2.icns file. This file contains the IV Swinger 2 icon in multiple resolutions and is in the Apple Icon Image format.

It also runs the fix_info_plist.py script after running PyInstaller. This script modifies the Info.plist file that PyInstaller generates to fix up a few things (version number, copyright string, resolution capability). It imports plistlib to assist with this.

The mac_run_pyi script now requires an argument "python" or "python3".

When the mac_run_pyi script has been run, there is a directory named "dist" containing the "IV Swinger 2.app" MacOS application bundle.

10.1.1.2 run_pyi.bat

The run_pyi.bat script uses the --icon option to point to the IV_Swinger2.ico file. This file contains the IV Swinger 2 icon in multiple resolutions and is in the Windows ICO file format. The --add_data option is also used to include that file in the application directory, which is needed for the Windows title bar icon (see *set_root_options()* method in IV_Swinger2_gui.py).

The run_pyi.bat script now requires an argument "python" or "python3".

When the run_pyi.bat script has been run, there is a directory named "dist", that contains an "IV Swinger 2" directory, which contains the "IV Swinger 2.exe" Windows executable.

10.1.2 Icon File Creation

The IV_Swinger2.icns and IV_Swinger2.ico files mentioned above were created as follows:

- PowerPoint was used to create the artwork. The basic drawing is a circle, filling a whole slide. 3D rotation was then used to tilt it back by 15 degrees, creating a slightly elliptical shape. The Apple app icon guidelines were useful.
- A PNG (rectangular) was exported from PowerPoint
- Preview (Mac app) was used to crop the rectangular PNG to an ellipse
- https://iconverticons.com/online was used to generate .icns and .ico files

10.1.3 Location of --add-data Files

The files added using the --add_data option are used by the IV_Swinger2_gui.py module and are in the same location as the executable. Finding the path to that directory is a bit tricky, but is described in a StackOverflow answer. The *get_app_dir()* global function in IV_Swinger2_gui.py returns the path to the application directory depending on whether the app is "frozen" (pyinstaller) or is being run from the command line with Python.

10.2 DMG file generation

The mac_build_dmg bash script uses the dmgbuild tool to create a Mac DMG installer. The dmgbuild_settings.py file is derived from the example in the documentation. The DMG_background.jpg file provides the instructions to the user for dragging and dropping the app to the Applications folder and opening it the first time.

10.3 MSI file generation

Creating the Windows installer is more complicated. The WiX toolset is used to create the MSI installer file.

The first step is to run the WiX heat tool:

```
heat dir ".\dist\IV Swinger 2" -ag -sfrag -sreg -template product -out heat.wxs
```

The resulting heat.wxs file is an XML file describing the attributes of the installation. It is very generic, with placeholders for many attributes.

The next step is to run The fix_heat_wxs.py script to customize the heat.wxs XML for IV Swinger 2. The script uses BeautifulSoup to parse and modify the XML and write it to an IV_Swinger2_<version>_win.wxs file. An article on Code Project was useful for creating this script. Refer to the script for details.

After that, the WiX candle tool is run to compile the customized .wxs file:

```
candle IV_Swinger2_*.wxs
```

The result is an IV_Swinger2_<version>_win.wixobj file.

The final step is to run the WiX light tool to link the .wixobj file:

```
light -b ".\dist\IV Swinger 2" -sice:ICE60 IV_Swinger2_*.wixobj
```

The result is the IV_Swinger2_<version>_win.msi installer file.

11 References

The list below is minimal. The document has many more references in the form of hyperlinks.

1. Perma-Proto PV Module (EMR) Instructable:
   https://www.instructables.com/id/IV-Swinger-2-a-50-IV-Curve-Tracer/
2. PCB PV Module (EMR) Instructable:
   https://www.instructables.com/id/IV-Swinger-2-PCB-PV-Module-EMR/
3. PCB PV Module (SSR) Instructable:
   https://www.instructables.com/id/IV-Swinger-2-PCB-PV-Module-SSR/
4. PCB PV Cell (EMR) Instructable:
   https://www.instructables.com/id/IV-Swinger-2-PCB-PV-Cell-EMR/
5. PCB PV Cell (SSR) Instructable:
   https://www.instructables.com/id/IV-Swinger-2-PCB-PV-Cell-SSR/
6. "IV Swinger: Design, Construction and Operation", Chris Satterlee,
   https://raw.githubusercontent.com/csatt/IV_Swinger/master/docs/IV_Swinger1/IV_Swinger_Design_and_Construction.pdf
7. IV Swinger GitHub repository:
   https://github.com/csatt/IV_Swinger
8. "Wireless IV Curve Tracer for long term field testing", Jason Alderman:
   http://jalderman.org/?p=57
9. Arduino:
   https://www.arduino.cc/
10. MCP3202 data sheet, Microchip Technology:
    http://ww1.microchip.com/downloads/en/devicedoc/21034d.pdf
11. TLV2462 data sheet, Texas Instruments:
    http://www.ti.com/lit/ds/symlink/tlv2462a.pdf
12. CPC1718 data sheet, IXYS:
    http://www.ixysic.com/home/pdfs.nsf/www/CPC1718.pdf/$file/CPC1718.pdf
13. Python 2.7:
    https://docs.python.org/2.7/
14. Python 3:
    https://docs.python.org/3/