Chapter 1. Configuration Hacks

They say the beginning is usually a good place to start anything, but this is a *Hacks* book. You're likely to skip around to things with interesting titles, or perhaps the one that starts in Old English because you were flipping through the pages and it looked like the book had some terrible printing errors.

That said, there are some things that it's nice to keep in the back of your head before you start building things, and the Raspberry Pi has a few unexpected quirks that it's good to be aware of. If you're entirely new to Linux, electronics hacking, or both, it's a good idea to give this chapter a read-through before proceeding with any of the other hacks. You just might learn something that will save your Pi (and the $35 of having to replace it).

Hack 1. Choose and Format the Right SD Card

The Raspberry Pi does not have any built-in flash storage; it needs an SD card to do anything. Picking the right one might seem simple, but we're here to help you make the right choice.

strengths, especially when it comes to education. A few factors should weigh into your card selection, though, and even if you think you've chosen well, you might still need to troubleshoot minor problems.

SD cards are sold with a class number (e.g., 4, 6, 10), in which a higher class number equates to a faster card. Most high-quality, Class-4-or-greater SDHC cards (i.e., a recognized name brand) should work for most purposes. Vendors that sell cards with a Linux distribution meant for the Raspberry Pi largely use SanDisk or Kingston brand SDHC Class 4 cards. You can find a thorough list of known, tested cards (as well as cards that don't work) at http://elinux.org/RPi_VerifiedPeripherals. That said, a faster card can as much as double your transfer rate (in terms of MB/sec), so if speed is critical to your use, you should go with a higher class card.

CLASS 10 TOO CLASSY?

The early Raspberry Pi firmware and bootloader didn't like Class 10 SD cards. This problem is supposed to have been fixed, but you'll still see people occasionally running into problems with Class 10 cards, so just be aware that it's a potential issue. If you have your heart set on a Class 10 card and the first one doesn't work, try a different brand. In addition, overclocking has been found to cause errors with Class 6 and Class 10 SD cards, regardless of size or brand, and the errors might not appear for a few days or weeks. Keep this in mind if you plan to overclock your Pi.

If decision making isn't your strong suit, you can also keep multiple cards around, each with a different purpose, for a single Raspberry Pi. If you'd like easy peace of mind, several vendors sell SD cards preloaded with Linux distributions for the Raspberry Pi, including a card containing NOOBS (New Out-Of-Box Software), which has several distro options on it. RS Components and element14 offer a card preloaded with NOOBS as an add-on when you purchase a Raspberry Pi.

lations and re-installations, as well as config file editing and web browsing (to research answers to boot problems) in a pre-boot environment. After all, this thing was designed for education, and you're not going to learn much if you can't even get started. It fits on a 4 GB card and gives you multiple choices about which distro you'd like to set up. After you've chosen, you can always return to the menu and make a different selection by holding down Shift during boot, either to try something new or to get a mulligan on a corrupted card. If you don't buy it on a preloaded card, you can download it from http://www.raspberrypi.org/downloads.

If you used one of the SD cards that's known to work and you're still having problems, you should check a few other things. Be sure that you've updated the firmware on the Pi (see [Hack #4]). If it was not a new SD card, be sure you fully formatted it first, and make sure you do so for the whole card and not just a partition.

First, find the card's device name:

```
$ su -c 'fdisk -ls'
```

or:

```
$ df -h
```

You're looking for something like `/dev/sdd` or `/dev/mmcblk0` with the size of your SD card. To format, run the `mkdosfs` command, replacing `/dev/mmcblk0` with the location of your card:

```
$ mkdosfs -I -F32 /dev/mmcblk0
```

This will make a single FAT formatted partition on the SD card. To be honest, it really doesn't matter very much how you format or partition the SD card in most cases, because when installing any of the system images for Raspberry Pi OS distributions that include partitions (such as Pidora or Raspbian), the partition table on the SD card will be completely overwritten by the installed OS image. The exception to that is NOOBS. By partitioning the disk with a single FAT partition, it is possible to

If you find that you have, say, an 8 GB card, and your computer thinks it's only 2 GB, you need to "grow" it to match. Or you might have found that your card's device name ends in `p1` (followed by `p2` and so forth):

```
/dev/mmcblk0p2          1.6G  1.5G   54M  97% /run/media/wwatson/rootfs
/dev/mmcblk0p1           50M   18M   33M  35% /run/media/wwatson/boot
```

This means your card is partitioned, and you should get down to one partition before formatting. Adjusting partitions and their sizes is most easily accomplished with a GUI tool called Gparted, a visual version of the command-line `parted`.

Hack 2. Mount the SD Card

While you can certainly access the files on the Raspberry Pi directly from within a running instance, mounting the SD card on a separate computer with an SD card reader makes many tasks (such as adding or editing files) easier.

The Raspberry Pi is a standalone Linux computer, but it really helps to have another computer on hand. In some cases, it might even be necessary. Fortunately, many computers now come with SD card readers built in, and if yours didn't, they're inexpensive and easy to come by. So, even if you buy your SD cards preloaded, you should probably still have an SD card reader and a second computer for interacting with your Raspberry Pi build.

Most Linux distributions for the Raspberry Pi create at least two partitions on the SD card. The first partition is always `/boot`, because the Raspberry Pi GPU reads its firmware from the beginning of the SD card. The second partition is usually `/` (also known as the *root partition*).

> Pidora labels the partitions on the SD card `boot` and `root`, but Raspbian does not use disk labels, so it is especially important to note the device names for that distribution.

mand should list all mounted partitions on the system. You are looking for something like `/dev/mmcblk0p1`, which means the first partition (`p1`) on the MMC block (`mmcblk`) device:

```
[spot@wolverine ~]$ mount
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
...
/dev/sda3 on / type ext4 (rw,relatime,data=ordered)
/dev/sda1 on /boot type ext4 (rw,relatime,data=ordered)
/dev/mmcblk0p1 on /run/media/spot/boot type vfat (rw,nosuid,nodev,relatime,
/dev/mmcblk0p2 on /run/media/spot/rootfs type ext4 (rw,nosuid,nodev,relatim
```

The last two lines in the output identify the MMC block device partitions mounted in `/run/media/spot/boot` and `/run/media/spot/rootfs`, respectively.

> Linux uses the term *MMC* to describe the driver for both MultiMediaCard and Secure Digital (SD) formats.

If your SD card is not mounted automatically, make sure it's inserted and look at the output from the `dmesg` command. You do not need to pass any options to `dmesg` (although piping it through `less` is always a good idea). When you run it, it will print out quite a bit of stuff, but the output is in order from the last time you have booted your Linux system.

You'll want to look at the end of the output. Specifically, you look toward the end of the output to figure out the name of the MMC block device. Figure 1-1 shows an example of the sort of messages you are looking for.

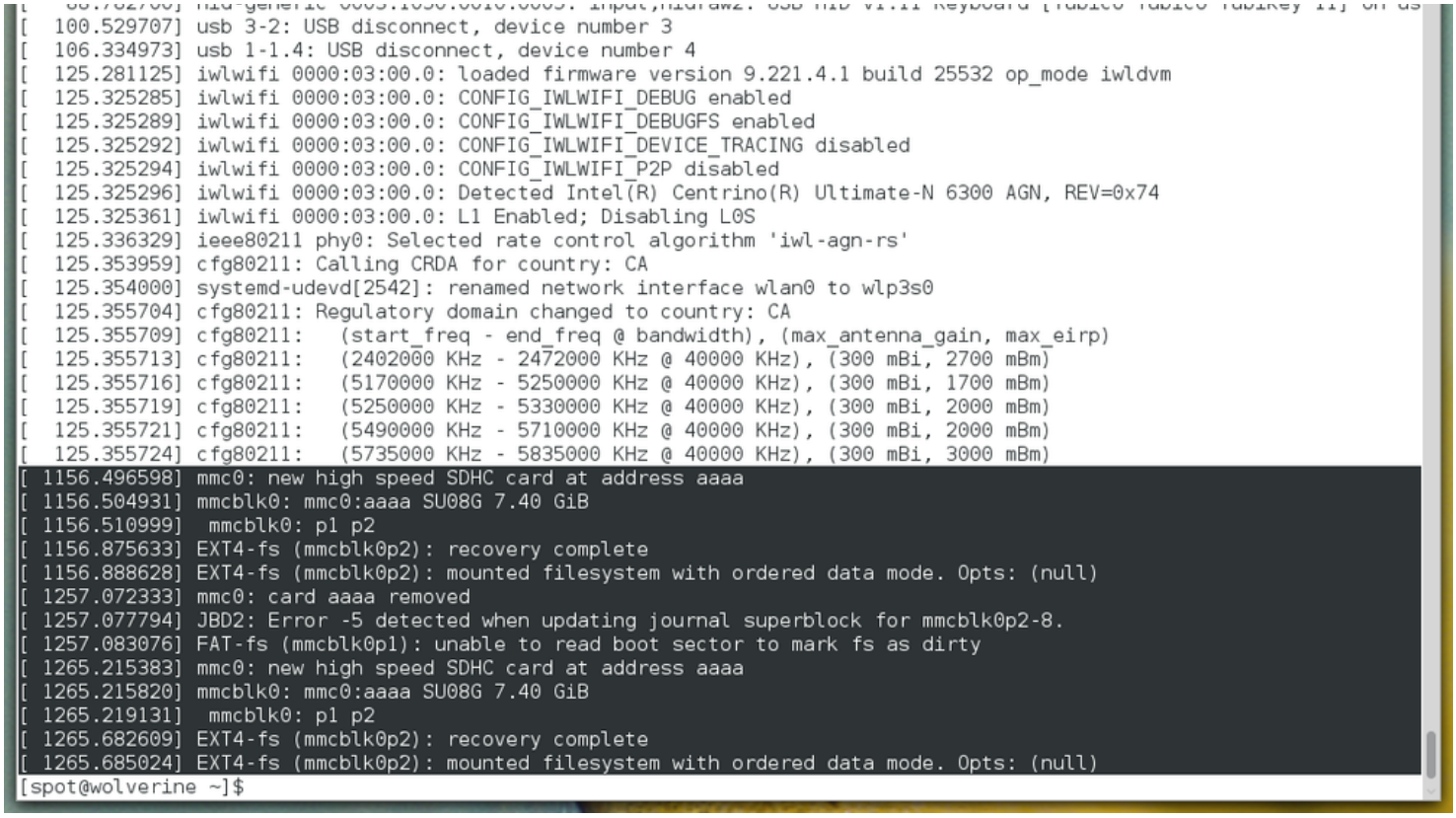

Figure 1-1. Output from dmesg on Fedora 19, with the MMC block device messages highlighted

In Figure 1-1, the MMC block device name is `mmcblk0`, and it has two partitions, `p0` and `p1`. This gives you enough information to determine the Linux device names for these partitions: `/dev/mmcblk0p0` and `/dev/mmcblk0p1`. You can confirm these are the correct device names by running:

```
brw-rw---- 1 root disk 179, 1 Aug 20 20:42 /dev/mmcblk0p1
brw-rw---- 1 root disk 179, 2 Aug 20 20:42 /dev/mmcblk0p2
```

If they exist, they're probably the ones you want (unless you have multiple SD cards inserted into your system somehow).

accidentally specify a hard disk drive instead.

Once you've identified the Linux device names for the MMC block device partitions on your system, you should be able to manually mount them by creating two mount point directories (as root):

```
$ su -c 'mkdir /mnt/raspi-boot'
$ su -c 'mkdir /mnt/raspi-root'
```

These directories will serve as anchors for mounting the partitions from the MMC block device.

Then, use the `mount` command to mount the boot and root partitions:

```
$ su -c 'mount /dev/mmcblk0p1 /mnt/raspi-boot'
$ su -c 'mount /dev/mmcblk0p2 /mnt/raspi-root'
```

If these `mount` commands return without errors, it means they have mounted successfully. You can confirm they have mounted by running `mount` again and piping the output through a grep for the MMC block device name (`mmcblk0`):

```
$ mount | grep mmcblk0
/dev/mmcblk0p1 on /mnt/raspi-boot type vfat (rw,relatime,fmask=0022,dmask=0
/dev/mmcblk0p2 on /mnt/raspi-root type ext4 (rw,relatime,data=ordered)
```

You should also now be able to see files in the `/mnt/raspi-boot` and `/mnt/raspi-root` directories.

It is also possible to mount the `/boot` partition inside the mounted `/` partition, but we recommend keeping them separate. That way, if you forget to mount the boot partition, it is more obvious, and you avoid the problem of accidentally copying files

Mounting the SD card is especially useful to make quick changes to the `config.txt` file that lives in the Raspberry Pi Linux `/boot` partition. If you need to change the ouput display settings for a new monitor (or an old HDMI TV with less than amusing quirks), it's a lot easier to do it from a mounted SD card than from a headless Raspberry Pi.

Just make sure the boot partition is mounted, and then change into that directory (`/mnt/raspi-boot`) and directly edit `config.txt` (as root). Save your changes, and then run `sync` to make sure the buffers get written back to the SD card.

When that finishes, change out of the directory (if you do not, Linux will not let you cleanly unmount the partition) and unmount both of the partitions (as root) with the `umount` command:

```
$ cd /mnt/raspi-boot/
$ su -c 'vi config.txt'
$ sync;sync;sync;
$ cd /mnt
$ su -c 'umount /mnt/raspi-boot'
$ su -c 'umount /mnt/raspi-root'
```

If the `umount` commands both return without any errors, it is now safe to remove your SD card. Just put it back in your Raspberry Pi, power it on, and hope for the best.

Hack 3. Decode the LEDs

Each Raspberry Pi has a set of LEDs in one corner that give you clues about what's happening (or not happening!) with the device. The Model A had only two lights, but the Model B offers a lot more insight and valuable troubleshooting information.

The Raspberry Pi Model B has five status LEDs (shown in Figure 1-2 and described in Table 1-1) that will help you troubleshoot problems when it won't boot or other

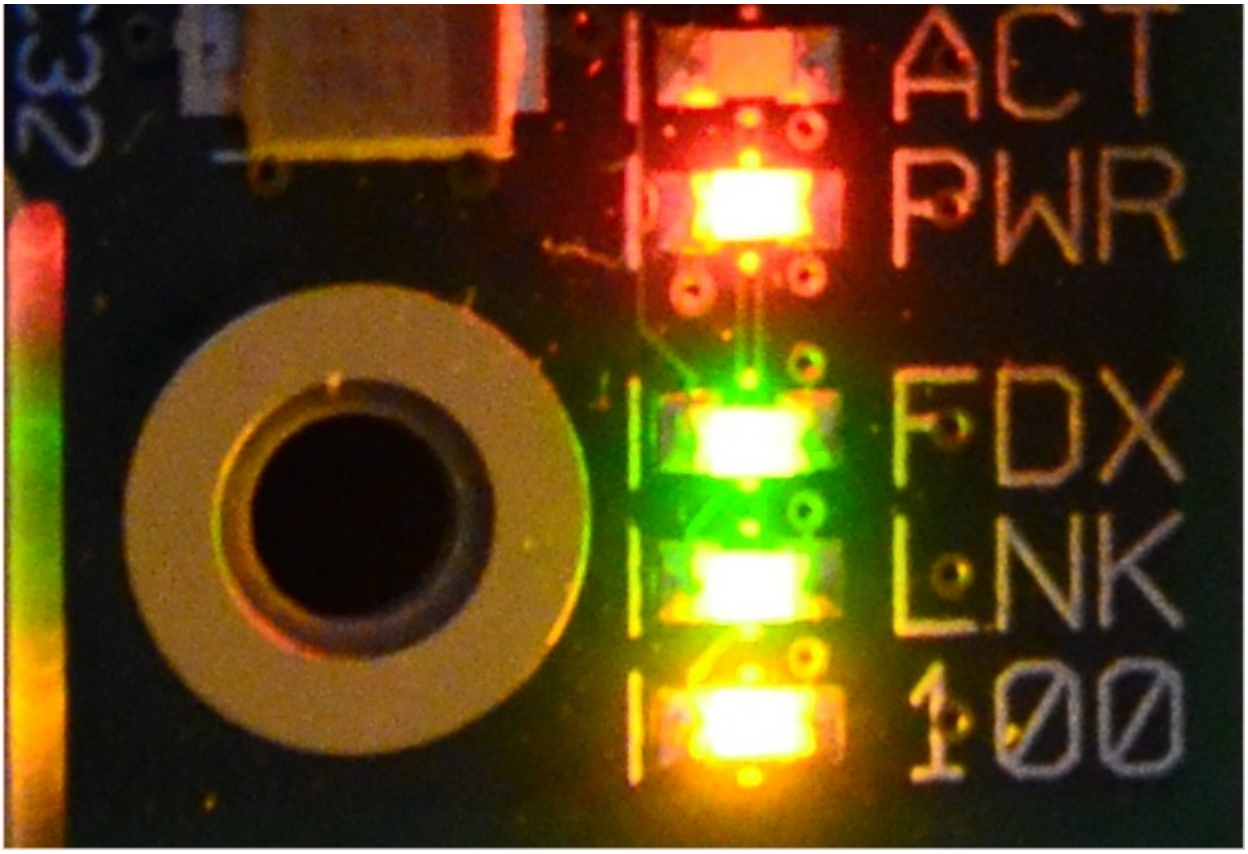

Figure 1-2. Model B LEDs

Table 1-1. Status LEDs on the Raspberry Pi Model B

Number	Label	Color	Function
D5	OK (Rev 1.0) ACT (Rev 2.0)	Green	SD card access, connected to GPIO 16
D6	PWR	Red	3.3 V Power, connected to 3.3 V
D7	FDX	Green	Full Duplex LAN
D8	LNK	Green	Link/Activity LAN
D9	10M (Rev 1.0) 100 (Rev 2.0)	Yellow	10/100Mbit LAN

Table 1-2. Common LED Error Codes

Light indication	Most likely problem
Red PWR light off	No power
Red PWR light on, green OK light off	The Pi can't read the image on the card. The voltage is below 5V.
Green OK light blinks 3 times*	start.elf was not found
Green OK light blinks 4 times*	start.elf did not launch
Green OK light blinks 7 times*	kernel.img was not found

Note that the flash patterns identified with an asterisk in Table 1-2 are accurate for the firmware available since October 20, 2012. Earlier firmware used different patterns, and later firmware may change these indicators as well.

The two files it's looking for, `start.elf` and `kernel.img`, absolutely must be on the boot partition. The first, `start.elf`, is the GPU binary firmware image, and `kernel.img`, as its name implies, is the Linux kernel. If the red PWR light is on, you know have power; then it's up to the green light to tell you what's gone wrong.

If the green light doesn't flash at all, the first thing you shuld do is check your SD card in another computer. Make sure that the image is written correctly. If all of the filenames look like somebody leaned on the keyboard, it did not write correctly! Format it and start again. If it does look OK, plug in nothing but the power and the

If the green light does blink, refer to Table 1-2 for information about what has gone wrong. Note that once `start.elf` has loaded, you'll see "the rainbow" (four large squares of color bleeding together). It should quickly go away as your Linux distro continues to boot, but if it doesn't, your problem is in the `kernel.img` file.

BLINK YOUR IP ADDRESS THROUGH THE LEDS

Pidora offers some features specifically for running in headless mode, including the use of the LEDs to communicate your IP address. See [Hack #11] to learn how.

Hack 4. Update the Firmware

The firmware your Raspberry Pi requires comes with any Linux distribution you choose, but it's frequently updated upstream, and your project might benefit from (or require) a more recent version.

The Raspberry Pi is a little different from your laptop, and even different from a lot of traditional embedded computers. The heart of the Raspberry Pi is the Broadcom BCM2835 system-on-chip, which is the CPU, GPU, and memory all combined in a single component. This detail is important, because the Raspberry Pi actually boots from the BCM2835 GPU. When you provide power to the Raspberry Pi, the CPU in the BCM2835 system-on-chip is actually disabled!

The Raspberry Pi boots like this:

1. **First-stage bootloader:** A bootloader programmed into the BCM2835 system-on-chip hardware mounts the FAT32 boot partition from the Linux distribution on the SD card. Note that this first-stage bootloader is programmed at manufacture time and is not modifiable or replaceable. A small, dedicated RISC core on the Raspberry Pi GPU starts this process.

2. **Second-stage bootloader:** Read off the boot partition on the SD card, this firmware (`bootcode.bin`) accesses the additional GPU firmware files,

CPU. An additional file, `fixup.dat`, configures the SDRAM partition between the GPU and the CPU. At this point, the CPU is released, and execution is transferred to it from the GPU.

4. **User code:** The CPU boots any supported binary, but the Linux kernel is the default. It assumes the filename is `kernel.img`, but you can be override the default in `config.txt`.

> Versions of the Raspberry Pi firmware prior to October 19, 2012 contained an additional third-stage bootloader (`loader.bin`), but this is no longer required or used. Previous builds also had different versions of the GPU firmware that had to be swapped in and out to enable different memory splits between the ARM CPU and GPU, but this is now configured in `config.txt`.

Because of how the Raspberry Pi boots, you must use an SD card to boot the Raspberry Pi; you cannot boot it from any other device (such as network or USB storage) alone. But this is a good thing. It prevents you from rendering the device unusable, because you cannot override the first-stage bootloader. If you end up with damaged, broken, or incomplete firmware, you can simply start over with a clean SD card.

The Raspberry Pi Foundation provides the firmware files that the GPU loads, which then enable the Raspberry Pi to boot a specially formatted Linux kernel image. All the Linux distribution images intended for use on the Raspberry Pi come with a copy of this firmware, but it is constantly updated upstream. To enable new functionality (or boot newer Linux kernels), you will want to make sure you are running the latest revision of the firmware.

The upstream home for the Raspberry Pi firmware is https://github.com/raspberrypi/firmware/. There is currently no source code available for these firmware files, so this repository contains only binary versions. Because the Raspberry Pi is so slow (especially for Git operations), we strongly recommend that you check out these files to your x86 laptop.

```
$ yum install git
```

or this command on Debian/Ubuntu:

```
$ apt-get install git-core
```

Next, create a working directory for Raspberry Pi related files, such as `~/raspi`:

```
$ mkdir ~/raspi
```

Go into the `raspi` directory:

```
$ cd ~/raspi
```

Use Git to get a local copy of the firmware files:

```
$ git clone https://github.com/raspberrypi/firmware.git
```

This will create a checkout in a new directory, named `firmware`. By default, this checks out the master branch, which at the time of this writing was synced up to the version of the firmware currently used by the Raspbian Linux kernel (3.2). If you are using a 3.2 kernel, this is the firmware you want to use. Another branch (named `next`) enables the updated drivers in the 3.6 Linux kernel. If you want to use this branch, change into the firmware directory and enter:

```
$ git checkout next
```

To switch back to the master branch, enter:

If you want to update your firmware again later, you don't need to check out this tree again. Simply go to the top-level checkout directory (`~/raspi/firmware`) and enter:

```
$ git pull
```

Remember, this will pull changes for the current branch only. If you want to pull changes for the other branch, you will need to switch to the other branch with the Git checkout command and run `git pull` there as well.

Now that you have checked out the repository and chosen your branch, your next step is to copy the boot firmware onto the SD card that has the Raspberry Pi Linux distribution image. To do this, you'll need to make sure the partitions on that SD card are properly mounted (covered in detail in [Hack #2]).

From here on, we will assume that the boot partition from your SD card with the Raspberry Pi Linux distribution image is mounted at `/mnt/raspbi-boot`. Current versions of Fedora (including Pidora) will automount it to `/run/media/$USERNAME/boot`, where `$USERNAME` is your username, so if you have it mounted somewhere else, substitute that mount point in the next set of instructions.

To update the firmware on the boot partition, all you need to do is copy the right files from the `firmware/boot` directory into the mounted boot partition (as root).

> You probably do not want to copy *all* of the files from this directory.

You're looking for these critical firmware files in the `firmware/boot` directory:

- `bootcode.bin`
- `fixup.dat`
- `start.elf`

```
$ su -c 'mv /mnt/raspi-boot/bootcode.bin /mnt/raspi-boot/bootcode.bin.backup'
$ su -c 'mv /mnt/raspi-boot/fixup.dat /mnt/raspi-boot/fixup.dat.backup'
$ su -c 'mv /mnt/raspi-boot/start.elf /mnt/raspi-boot/start.elf.backup'
```

Copy each of these firmware files (as root) into the mounted boot partition:

```
$ cd ~/raspi/firmware/boot/
$ su -c 'cp -a bootcode.bin fixup.dat start.elf /mnt/raspi-boot/'
```

SU VERSUS SUDO

The command example used here for copying firmware files (along with most other command examples in this book) use `su`. The `su` command will prompt you for the root password of your Linux laptop. If you have configured `sudo` for use on your Linux laptop, you can replace the `su -c` command with `sudo` and the command to copy the firmware files (as root) will look like this instead:

```
$ sudo cp -a bootcode.bin fixup.dat start.elf /mnt/raspi-boot/
```

Whichever method you prefer is fine, as they are both valid methods for Linux operations as the root user. We use the `su -c` syntax in most examples throughout the book for all root operations because it will work in all cases, whereas `sudo` works only if it is configured for your user on that Linux distribution. If you an encounter instructions prefaced with `sudo`, know that `su` is an option when you don't have `sudo` configured.

When the new Raspberry Pi firmware finishes copying onto the boot partition, run the `sync` command to ensure the data has all arrived onto the SD card:

```
$ sync
```

Then it should be safe to unmount the SD card partition(s) and eject the SD card. You can unmount these partitions from the GUI interface of your Linux laptop, or you can manually unmount them from the terminal by changing into a directory that is not in either of the mounted partitions and then enter:

At this point, the SD card will contain the new firmware. You'll know that the update worked if the Raspberry Pi still boots into the Linux image, but at a minimum, the firmware will draw a multicolored "rainbow" box (see "Somewhere Over the Rainbow..." sidebar) to the configured output device (usually an HDMI connected one) as its first step in the boot process (unless you have explicitly disabled this behavior in `config.txt`). If that occurs, the firmware is properly installed onto the SD card.

SOMEWHERE OVER THE RAINBOW...

Hopefully, if everything goes well with your Raspberry Pi, you'll never have to see the "rainbow" screen (shown in Figure 1-3) for more than a fraction of a second when it boots up. The screen is generated by the Raspberry Pi firmware as it initializes the GPU component of the BCM2835 system-on-chip.

To test that the output works successfully, the GPU draws four pixels on the screen and then scales those pixels to be very large, resulting in the multicolor screen. If your Raspberry Pi ever refuses to go over the rainbow and into a proper Linux boot, it means that the configured Linux kernel image (default: `kernel.img`) was not able to boot.

Figure 1-3. The "rainbow" screen (uploaded to http://elinux.org/File:Debug-screen.jpg by user Popcornmix and shared under the terms of the Creative Commons Attribution-ShareAlike 3.0 Unported License)

Some optional versions exist for some of the Raspberry Pi firmware files. It is possible to configure the Raspberry Pi to dedicate the minimum amount of memory to the GPU (16 MB). When this is done, the Raspberry Pi Second Stage Bootloader looks for `start_cd.elf` and `fixup_cd.dat` instead of `start.elf` and `fixup.dat`. [Hack #24] provides a longer discussion on GPU/CPU memory splitting.

how to point a webcam at it and stream the video to the Internet. Just kidding! These tools can monitor the physical state of your tiny hardware.

A “normal” Linux computer would likely include onboard health monitoring sensors. Quite a few monitoring chips and components are used in various systems, but on the Raspberry Pi, all of that hardware is entirely hidden inside the Broadcom system-on-chip, so you can’t access it with those usual methods.

To reach those components to monitor your Pi’s health, you need to use the `vcgencmd` utility. It should be preinstalled with any of the general-purpose Raspberry Pi Linux distributions available, but if it’s not, you can get a copy from the firmware tree at https://github.com/raspberrypi. If your distribution is compiled for ARM hardware floating point, look in the `hardfp/` subdirectory; otherwise, look in the `opt/` subdirectory.

and RaspBMC) are built for the ARMv6 hard-float architecture, because that gives the best possible performance on the Raspberry Pi. However, some older releases of these targets (and other OS platforms) were built with optimization for ARMv6 soft-float. These two optimization levels are not compatible with each other. On Linux, there is a good way to check for support for ARMv6 hard-float, using the `readelf` command:

```
$ readelf -a /usr/lib/libc.so.6  | grep FP
```

You can run this command directly on the Raspberry Pi Linux distribution (you might need to install the `elfutils` package first), or you can copy a binary or library from within the Raspberry Pi Linux distribution and onto another system with `readelf` handy.

If the binary has support for ARMv6 hard-float optimization, you will get output that looks like this:

```
Tag_FP_arch: VFPv2
Tag_ABI_FP_rounding: Needed
Tag_ABI_FP_denormal: Needed
Tag_ABI_FP_exceptions: Needed
Tag_ABI_FP_number_model: IEEE 754
Tag_ABI_HardFP_use: SP and DP
Tag_ABI_VFP_args: VFP registers
```

The important line is the last one, `Tag_ABI_VFP_args: VFP registers`. It will show up only if the binary being checked is built with ARMv6 hard-float optimization.

Once you've installed it (if necessary), look at the options that `vcgencmd` offers:

```
$ vcgencmd commands
```

This will output a list of all the commands that you can pass to the `vcgencmd` tool:

```
commands="vcos, ap_output_control, ap_output_post_processing, vchi_test_init,
```

Unfortunately, it doesn't actually tell you anything about those commands or what they do. Some of them seem obvious, but then when you run them, they return things like this:

The tool is poorly documented, but the Raspberry Pi community has come together and figured some of them out.

Measure Component Voltage

The `vcgencmd measure_volts` command shows the voltage for some of the key Raspberry Pi components, specifically:

`core`
: The GPU processor core

`sdram_c`
: The SDRAM controller

`sdram_i`
: The SDRAM input/output (I/O)

`sdram_p`
: The SDRAM physical memory

Each of these components can be passed as an option to the `vcgencmd measure_volts` command (if you don't specify one, it will return the value for `core`).

You might be wondering why you'd care about measuring these voltages, and in most cases, you probably don't. They'll sit happily as shown in Table 1-3.

core	1.20
sdram_c	1.20
sdram_i	1.20
sdram_p	1.23

The only time you might care about the component voltages is if you decide you want to overclock your Raspberry Pi. All of these voltages are configurable (as covered in detail in [Hack #6]).

You might expect this command to return the system board voltage (which varies between 4.75V and 5.25V under normal conditions), but it doesn't. See [Hack #9] for how to do that manually.

Measure Temperature

The `vcgencmd measure_temp` command reports the core temperature of the BCM2835 system-on-chip on your Raspberry Pi (in Celsius):

```
temp=44.4'C
```

Alternatively, you can get the same temperature reading by reading this value directly from `/sys/class/thermal/thermal_zone0/temp`:

```
$ cat /sys/class/thermal/thermal_zone0/temp
44388
```

Celsius temperature by 1.8 and add 32. To get Kelvin, add 273.15 to the Celsius temperature. Is it getting hot in here, or is it just us?

From the perspective of monitoring the Raspberry Pi hardware, this reading is probably sufficient. Since there is really no separation of the CPU/GPU (at least not from a physical or heating perspective), this gives you an idea of how hot the board is running. That said, if you want a more detailed (or just an additional) temperature reading, you can wire in an additional temperature sensor, as described in [Hack #37].

Monitor Memory Split

Whether hardcoded or dynamically allocated, the `vcgencmd get_mem` command returns the value for either the ARM CPU or the video GPU.

To see the amount of memory currently split off for the ARM CPU, run:

```
$ su -c 'vcgencmd get_mem arm'
arm=448M
```

To see the amount of memory currently split to the video GPU, run:

```
$ su -c 'vcgencmd get_mem gpu'
gpu=64M
```

Check Custom Configuration Overrides

Have you forgotten what configuration changes you have made to your Raspberry Pi? Specifically, the ones that change settings in the firmware? While you could look in `/boot/config.txt`, the `vcgencmd get_config` command is here to help you.

To see all configurations with a number (integer) datatype, run:

To see all configurations with a text (string) datatype, run:

```
$ su -c 'vcgencmd get_config str'
```

> There are very, very few configuration options that store string values instead of integers. Don't be too surprised if the `vcgencmd get_config str` command doesn't return anything.

If you just want to check the value of a specific configuration, pass that config name instead:

```
$ su -c 'vcgencmd get_config arm_freq'
arm_freq=900
```

The `vcgencmd` utility is not the most user-friendly tool, but it does have a deep connection into the inner workings of the Raspberry Pi. Since this tool is open source (and the source code is available in the aforementioned Raspberry Pi GitHub firmware checkout), if you want to go very deep into the inner workings of the Raspberry Pi hardware, looking at the `vcgencmd` source code is a good jumping-off point.

Hack 6. Overclock Your Pi

The Raspberry Pi is not a notably fast computer. For most projects, it is more than capable of providing enough performance to get the job done, but for other projects, you might want to overclock the hardware to get a little bit more horsepower.

The Raspberry Pi hardware is preconfigured to what the manufacturer believes is the best balance of reliability and performance. Now that we've stated that for the record, it also comes with a lot of tuning knobs, and if you are feeling brave, you can turn them up to get extra performance out of the hardware.

the same Intel CPU was whether it passed speed tests. The ones that passed got labeled at the higher clock speed, while the rest got the lower clock speed. If you were lucky, you could adjust settings to get a higher clock speed.

These days, overclocking refers to changing any sort of setting to get performance above and beyond the default configuration of the hardware. As an example, some people have resorted to any number of tricks and hacks to get a performance boost, including immersing the entire system in liquid nitrogen cooled Flourinert. Some people are crazy.

> This is an excellent time to warn you: trying to overclock your Raspberry Pi will almost certainly make the hardware burn out quicker, possibly immediately. It will also probably not double your performance, and if by some miracle it did, you probably wouldn't be able to run anything reliably on the overclocked Raspberry Pi.
>
> Then again, this is a $35 PC. You live only once. (When you decide to really take that advice to heart, try [Hack #40].)

Remember that the heart of the Raspberry Pi is a Broadcom system-on-chip, with an ARM CPU, a Videocore IV GPU, and 512 MB of RAM. Each of these parts have its own clock frequencies, and the GPU has adjustable clock frequencies for its subcomponents. Specifically, the GPU has a core frequency, an H264 frequency (the H264 hardware video decoder block), a 3D processor frequency, and an image sensor processor frequency.

You can tweak all of these settings by changing options in `/boot/config.txt`. This file may or may not exist; if it does not, just create a new empty file.

Increase ARM CPU Frequency

Let's start with the most obvious overclock: the ARM CPU. The frequency of the ARM CPU (`arm_freq`) defaults to 700 MHz. To speed it to 900 MHz, add this line to `/boot/config.txt`:

Then, when you reboot, the hardware will try its best to honor your request. But remember, this isn't magic. No matter how badly you want to put `30000000` as the frequency, it isn't going to work. People with a lot of experience overclocking hardware have determined that the Raspberry Pi does not usually successfully overclock beyond 900 MHz, unless you use *overvolting* (see [Hack #7]).

Increase SDRAM Frequency

Another simple way to overclock is to increase the frequency of the SDRAM memory. The frequency of the SDRAM memory (`sdram_freq`) defaults to 400 MHz. You can usually increase this value to 500 Mhz without issue by adding this line to `/boot/config.txt`:

```
sdram_freq=500
```

Just like with `arm_freq`, you'll need to reboot your Raspberry Pi for this to take effect.

Increase GPU Frequency

Your last major overclocking option is the GPU components, the frequencies of which are all defined by `gpu_freq` and default to 250 MHz.

`gpu_freq` is a sort of *super setting*. Setting it assigns the same value to the `core_freq` (GPU processor core frequency), `h264_freq` (hardware video block frequency), `isp_freq` (image sensor pipeline block frequency), and `v3d_freq` (3D block frequency). If you have a GPU-intensive task, you might get some extra performance by increasing the `gpu_freq` to 325. You can do this by adding this line to `/boot/config.txt`:

```
gpu_freq=325
```

(listed previously) either the same or different by a factor of an integer multiplier. If you do not do this, the GPU components will receive a mixture of incompatible pulses and things will stop working very quickly.

However, because the `core_freq` value also includes the L2 cache and some of the SDRAM memory clock cycles, increasing just that value could give the ARM CPU a performance boost. Multiply the default value by 2 (the largest integer that will really work) and set the value to 500 in `/boot/config.txt` like this:

```
core_freq=500
```

> Note that this might not work. Some people report success, while others report failure. If you try to mix this `core_freq` change in with the other overclocking features, it might work only when they are set low (or left at the default).
>
> We cannot emphasize this enough: sometimes, when overclocking fails, it does so in less-than-obvious ways. Reliable programs become buggy, hardware devices stop working at random, and the system might just reboot for no good reason.

When you do overclock, you'll want to have a quantifiable test case that you can run over and over again to see what gives you the best performance for the workload that you care about on your specific Raspberry Pi. Do not simply download a canned benchmark and trust it. A benchmark designed to show GPU performance will not help you optimize your overclocked Raspberry Pi system for tasks that are CPU-bound.

Pretested Overclock Presets

Newer versions of the Raspberry Pi firmware contain the option to choose between five overclock (*turbo*) presets that try to get the most performance out of the SoC without impairing the lifetime of the Pi. This is done by monitoring the core temperature of the chip and the CPU load and dynamically adjusting clock speeds and the core voltage.

pending on the individual board and which of the turbo settings is used. Table 1-4 details the current settings of the five overclock presets.

Table 1-4. Overclock presets

Preset	ARM	Core	SDRAM	Overvolt
None	700	250	400	0
Modest	800	250	400	0
Medium	900	250	450	2
High	950	250	450	6
Turbo	1000	500	600	6

If you are running a current version of Raspbian, you will notice that the `raspi-config` tool has support for configuring your Pi into any of these five presets. For other distributions, you will need to define the preset you want to use in `/boot/config.txt` by passing the values for each option individually. For example, to set the Medium preset, add these lines to your `/boot/config.txt`:

```
arm_freq=900
core_freq=250
sdram_freq=450
over_voltage=2
```

Also, just because the Turbo setting has been known to work with some Raspberry Pi units, that doesn't mean it will work with yours. Quite a few users have reported SD card corruption when trying to run their Raspberry Pi at that overclock preset.

to get more performance out of an electrical component.

The circuits in your Raspberry Pi are made up of transistors that act as logic gates or switches. The voltage at these nodes switches between a high voltage and a low voltage during normal operation. When the switch changes, the capacitance of the transistor and the voltage applied affect how quickly the switch output changes. Configuring a circuit to use higher voltage ("overvolting") allows the circuit to react faster, which permits you to overclock the hardware further than what would normally be possible.

The Raspberry Pi firmware exposes some configurable voltages, which map up with the following values in `/boot/config.txt`:

- `over_voltage` (core)
- `over_voltage_sdram_c`
- `over_voltage_sdram_i`
- `over_voltage_sdram_p`

> If you do *overvolt* your Raspberry Pi by changing any of these settings, it might permanently set a fuse in your BCM2805 system on chip. That means that the vendor will know if you overvolt the hardware, it burns out, and you try to return it as defective. We shouldn't have to say that it's not OK to return things as defective when you were responsible, but you should be aware that this is warranty-voiding behavior.

The biggest change comes from adjusting the `over_voltage` value, which is the core voltage for the ARM CPU and GPU in the BCM2835. The possible values for `over_voltage` run from -16 (0.8 V) to 8 (1.4 V), with default value at 0 (1.2 V). Each integer above (or below) 0 steps the voltage by 0.025 V. You cannot go over 6 without also setting `force_turbo=1` (note that this will probably trip the "warranty voided fuse").

(SDRAM physical voltage) settings. It is possible to set those settings independently, but you are far more likely to get them wrong (or mismatched) and end up with memory corruption, so we strongly recommend that you use the `over_voltate` super-setting instead.

If you decide to overvolt, just set these configuration options in `/boot/config.txt`, and then reboot.

When you're overvolting (or overclocking as well), monitoring the voltage levels of the components you've bumped up suddenly makes more sense. These methods can nudge out a tiny bit more performance from the hardware, but you're trading that extra bit of performance for a reduction in hardware lifetime (and possibly stability as well).

Hack 8. Get More USB Ports

The Raspberry Pi Model B has two dedicated USB connector ports, but really, that just isn't enough for an awful lot of use cases. Here's how you can hack in a few more.

Universal Standard Bus (USB) has become the de facto standard connector for computing accessories. Keyboards, mice, hard drives, joysticks, flashlights, and even foam missile launchers all connect via USB. The Raspberry Pi (Model B) comes with two dedicated USB 2.0 ports to allow you access to this wide world of peripheral goodness, but these ports get used up quickly. The normal use case of a keyboard and mouse will use up both of these connectors, and you're left with no place to put anything else!

This is not a new problem for computer users. Laptops usually come with one to three USB connectors as well, even though a single USB host controller can support many more devices running simultaneously on the same BUS (up to 127 devices, to be precise). The trick to getting more is to use a USB hub.

Bus powered

This type of USB hub draws all its power from the host computer's USB interface and is the type you're likely to acquire as a free giveaway or in the cheap-stuff bin at the electronics store.

Externally powered

Also known as *self-powered*, this type of USB hub has an external power supply and uses it to provide full power to each USB connector on the hub.

USB 2.0 current is allocated in units of 100 mA (called *unit loads*), up to a maximum total of 500 mA per port. This means that if you are using a bus-powered hub, in the best possible scenario (getting 500 mA from the host computer), it can power four devices. That's what the specification says, so it must be true, right? But in the real world, this isn't quite the case.

For starters, the USB hub needs some power to run, so it won't be able to take the 500 mA from the host computer and give it all to the ports. Even if we assume it is an extremely efficient device (they usually are not), that means it can provide one unit load to four devices at once. But that's not the whole story.

The USB specification is pretty loose as specifications go (partially as a result of its ubiquity), and lots and lots of devices want more than 100 mA to work properly—most notably, wireless networking USB devices and keyboards with fancy features (LCD displays, integrated USB hubs, backlights, blenders, etc.). These devices are classified as *high-power* USB devices and can use up to the maximum of five unit loads (500 mA) per port. They are rarely (if ever) labeled as such, and they look visually identical to low-power (single-unit load) devices.

On top of all that, the dedicated USB connectors on the Raspberry Pi provide only one unit load (100 mA) per port instead of the five unit loads that a "normal" computer would. This amount isn't nearly enough to power a bus-powered hub with any-

This is why if you connect a high-power USB device directly to the Raspberry Pi, it will either attempt to operate in low-power mode (sometimes these devices can do that), or the Raspberry Pi will simply power off or refuse to see the device. The majority of high-power devices will detect at low power, then try to pull additional power when put into active use (this is particularly common with wireless devices), resulting in a confusing scenario where the device appears to work, and the Linux kernel drivers load, but it doesn't actually work reliably or properly.

The solution to this problem space for the Raspberry Pi is to use an externally powered USB hub. You will want to use a good one, though, because there are plenty of awful choices here as well. It is common for the manufacturers of these USB hubs to cut corners and design the hub to run off of a low-amperage power supply. They do this because they assume that most of the devices you will connect to it are low powered and that you will not have all of the ports used at once.

It is not uncommon for inexpensive, seven-port hubs to use a 1 A power supply. If each of those seven ports is connected to a high-power (five unit loads, 500 mA) device, they would need a 3.5 A power supply. More, really, because the hub needs power too!

To be safe, you should assume the opposite from what these cost-cutting manufacturers do. Just assume that any USB device you want to connect to your Raspberry Pi is high powered and that each port in your USB hub will have a high-powered device connected to it. Then it is a simple math problem to confirm if a USB hub will be a good choice:

1. Take the number of ports on the USB hub, and add 1 (to account for the USB hub itself).

2. Multiply that number by the size of a high-power load (.5).

The result will be the number of amps that the power supply for your USB hub should be providing (at a minimum).

backpower.

The standard says that hubs aren't supposed to do this, but plenty of them do. Backpower can result in a situation where the connected USB hub has power before the Raspberry Pi has power (across the standard micro-USB power connector), which would cause the Raspberry Pi to be in a partially powered-on state. While partially powered on, your Raspberry Pi might start to make unwanted writes to the SD card.

To avoid this, you can plug the USB hub's power supply and the power supply for your Raspberry Pi into the same power strip, then use the switch on the power strip to power them on simultaneously.

The Pi Hut sells a seven-port USB hub designed specifically to be ideal for the Raspberry Pi. It avoids the need for careful power-on ordering, because it will never feed any power back over the interconnect cable. Sadly, however, it has only a 2 A power supply, which means you can have high-power devices (using five unit loads) on only three ports at once, with the leftover power going to the hub. Still, this unit is designed not to backpower, so you'll never have to worry about that.

There is also a four-port hub that is known to not have backpower issues. Even though it also has a 2 A power supply, you're arguably less likely to exceed that on a four-port USB hub than you would be on a seven-port USB hub.

The best hub for the Raspberry Pi that we've seen so far is the PIHUB. It is a four-port externally powered hub with a 3 A power supply, and it is in the shape of the Raspberry Pi logo. They don't have a U.S. version at the time of this writing, but they say it is coming soon!

Hack 9. Troubleshoot Power Problems

The Pi doesn't need a lot of power, but that also means that it needs what it's asking for, and you can run into trouble when it gets too much or too little.

If you have a charger for most Android phones, you have the Pi's power cable (sorry, iPhone fans). It is possible (but not the best scenario and might not work at all) to plug the other end into the USB port of your computer rather than the wall. And for other projects, you'll want to get power through the GPIO. That said…

Think Twice Before Using the GPIO to Power the Pi

Before you rush to input 5 V over the GPIO pin, remember that when you do this you're bypassing the hardware's input fuse. That fuse is there to protect your hardware in case of malice or stupidity.

For example, imagine that you think you're passing 5 V, but you're actually passing more than that into the Raspberry Pi via the GPIO. That might be because you weren't entirely clear on what you were doing, or it could just be an accident. Either way, out comes magic smoke! And by "magic," we mean, "that project just disappeared like a bunny in a hat!"

Plenty of power supplies aren't perfectly "clean," meaning it might *say* "5 V," but what it means is "more or less 5 Vish." Even if it just spikes above 5 V, you're bypassing the transient-voltage-suppression (TVS) diode!

That diode is what would normally protect the Raspberry Pi from those unexpected voltage spikes (by shunting away the excess current when it exceeds the avalanche breakdown potential), but you're just going right around it. And then out comes the magic smoke.

Last, but not least, you have to put *regulated* 5 V into the GPIO, and most power adapters do not output regulated voltage. This means you need to have a voltage regulator circuit between the GPIO pin and the power adapter.

For all of these reasons, we highly recommend you just feed power into the Micro USB Type B port, unless you have a truly excellent reason not to.

which can actually partially power the Raspberry Pi. Do not do this.

USB hubs are not supposed to backpower. This is not regulated or reliable power in any real sense. It can (and likely will) result in unpredictable behavior including (but not limited to) unexpected program failures, kernel panics, and SD card corruption.

I STILL WANT TO BACKPOWER THE PI!

OK, fine. There is a reasonably reliable way to do this. Some industrious hackers in Australia had a custom USB 3.0 hub produced with the explicit purpose of providing backpower for a Raspberry Pi.

Specifically, the interconnect port on their hub will send over the 1000 mA (1 A) that the Raspberry Pi needs for normal operation. This will power the Pi entirely off the connection to the USB hub (no separate power source is necessary). You can check it out here:

http://www.buyraspberrypi.com.au/shop/4-port-usb-3-0-powered-usb-hub/

The only downside to using this hub is that it has only a 2 A power supply, and 1 A is going to the Raspberry Pi, leaving a little less than 1 A (some of that needs to go to the hub itself) for the connected devices. That doesn't leave a lot of room for too many high-power devices (.5 A at maximum five-unit load).

Get Power Through GPIO Safely

Note that there is a big difference between using the Raspberry Pi GPIO pins to power an attached device and pushing 5 V into the GPIO to power the Raspberry Pi. Lots of the hacks in this book need to draw some current from the GPIO pins, and this is safe to do.

There are 3.3 V pins (P1-01 and P1-17), in addition to the 5 V pins (P1-02 and P1-04). Maximum permitted current draw from the 3.3 V pins is 50 mA. Maximum permitted current draw from the 5 V pins varies between the Raspberry Pi Model A and Model B hardware. The value for the maximum permitted current draw from the 5 V pins is calculated by starting with the USB input current (nominally 1 A), then subtracting the current draw from the rest of the board.

On the Model A, the board has a current draw of 500 mA, so the max current draw off the 5 V pin is 500 mA. On the Model B, because it has a higher current draw on

other GPIO P1 pins, you're likely to fry the entire Raspberry Pi. While deep-fried raspberry pie sounds like a delightful carnival snack, a fried Raspberry Pi circuit board is neither tasty or desirable.

Solve Power Problems

Now that you've reviewed your power options, you have to figure out what to do when things go awry.

When the power is too low (or the current is too low), the Raspberry Pi starts to act… well, the technical term is "weird." USB devices might not show up, or they might blink in and out randomly. Software might not run reliably. Cats and dogs living together, mass hysteria!

If things are just being "weird," there's a good chance insufficient power is to blame. There are two main reasons why this can happen, even when you think you've done everything right: a subpar power supply or a faulty Micro USB cable.

Get a better power supply

It's the opposite of that power spiking problem mentioned in Think Twice Before Using the GPIO to Power the Pi. It *says* "5 V," but what it means is, "I might consider delivering 5 V on my best day when all the stars are aligned and you perfectly hum the *Doctor Who* theme song backward to appease my cranky nature." But they couldn't fit all those words on the plug, so they just put "5 V." (Or at least that's our theory.)

Sadly, this is a common scenario. A lot of cheap Micro USB cell phone chargers are cheap for a good reason: they don't work very well. (Did you buy it at a dollar store? That could be a clue it's a cheap one.)

When it's for your phone, it's no big deal. It just takes longer to charge your battery. But the Raspberry Pi won't take so kindly to the drop in desired power.

power supply and the Raspberry Pi.

Get a better micro USB cable

Your Micro USB cable is less likely to be subpar in quality than the power supply, but it does happen. Cables have conductors in them that provide some resistance, but that's usually irrelevant.

For example, at 5 ohms and 50 mA of current, the voltage drop across the cable might be about 250 mV. Most devices are OK with that, because the USB specifications require that they be tolerant of voltage drops of that amount. However, some devices, like the Raspberry Pi, want more power (especially if you have some hungry USB devices plugged directly into the Raspberry Pi's USB ports).

As mentioned previously in this hack, the Model B draws a peak current of 700 mA, so if your USB cable has 5 ohms of resistance, it would result in a 3.5 V voltage drop. As far as the Pi is concerned, that's huge.

The good news is that most USB cables don't have 5 ohms of resistance, and the really good ones will be very close to 0. We haven't really had problems with cables that came with modern phones, which seems to be most people's source of such cables.

If you need to purchase a USB Micro B cable, Mediabridge's "USB charging cables" test with a low resistance and are available on Amazon. Adafruit's USB Micro B cables also work fine in peak-current draw on the Model B.

WHY DO USB DEVICES CAUSE MY PI TO REBOOT?

Hotplugging (plugging in a USB device when the Pi is already running) will often cause the Pi to reboot. Plugging in the device causes a power spike, which means a drop in power to the Pi, which leads to the reboot.

The Rev 1 board had two 140 mA polyfuses on the USB ports that prevented this but caused other problems, so Rev 2 boards do not. If you do anticipate the need to hotplug a device, do it through a powered USB hub.

But if you think you have a problem with your cable and want to test its resistance, you can either take the cable apart or you can use an accessible Micro USB Type B device (something that has ground pins on it).

The USB Type A connector is big enough that you can get to the ground pin directly. (It's Pin 4, the first pin on the left if you're looking down the cable with the hollow space at the top of the connector.) Measure resistance with a calibrated multimeter set to the lowest ohm setting from ground on the Micro USB-B connected device to ground on the USB Type A connector to get a good idea of the cable's resistance.

You can also get a little hardware tool to simplify this. Bitwizard B.V. makes a USB prodder just for this purpose.

Some people have even made their own cables by soldering low-resistance wires to a power supply and a Micro USB Type B connector. This is a neat hack if you just want to try making cables, but it's not really necessary.

Most modern USB cables do not seem to have serious resistance issues. If it will charge a Micro USB Type B cell phone quickly and reliably, it is probably good enough for the Raspberry Pi. If it won't, a new one is generally easy and cheap to come by.

Hack 10. Unbreak Your Raspberry Pi

The Raspberry Pi hardware is pretty rugged for its size, but it does have one notable weak point that you might discover. Here's how to find it and how to hack it back to life if it breaks.

The Raspberry Pi comes with a built-in self-destruct button that many people have accidentally triggered the first time they plugged it in. OK, that's not *precisely* true. But the placement of one of the Pi's fragile components makes it really easy to destroy your new toy before you've gotten to play with it. Here's what to do in case you broke it before you got around to reading this hack.

really good spot to grip when you're plugging in or unplugging your micro USB cable. It's not. Don't touch it. It's not a *critical* component, and your Pi *could* still work without it, but it also might not.

Figure 1-4. C6 is the black and silver cylinder beside the power connector

> If you're looking for a replacement in online stores, try searching with the terms "220 uF 16v electrolytic capacitor."

The relative fragility of this piece's connection is one of several good reasons to make or buy a good case for your Raspberry Pi. Meanwhile, if you need to carry it around, use the original static bag and box it came in.

tion in the knowledge that you're not the only one, and though it's not covered under warranty, you have a few options. First, a new and better power supply might fix the problem. With a stable power supply, you shouldn't have any problems.

If you don't know whether your power supply is doing what it should (beyond the obvious evidence of "it works" or "it doesn't work"), you can test it. The first sign that you're not getting consistent or strong enough power is not an *unworking* Pi, but rather an *unreliable* one.

When things start acting up mid-stream—all was fine at first, then maybe when you get into the GUI, the peripherals stop working—that's when it's time to check your voltage. The Pi has test points labeled TP1 and TP2 to help you. TP1 is just under the Raspberry Pi logo, and TP2 is between the GPIO and the RCA out.

Set your multimeter to 20V in the DC range and touch the leads to TP1 and TP2. The reading should be near 5 volts and certainly no more than 0.25 volts away in either direction.

Replace the C6 Capacitor

If your Pi is still not working or if you just want the capacitor back on there, you can solder a new one on.

> Soldering on a new capacitor *might* make your Raspberry Pi as good as new. It *definitely* will void your Raspberry Pi warranty.

When you solder it back on, note that the capacitor is polarized, and thus it is critical to have the black stripe facing the edge of the board. For some basic soldering tips, check out Soldering Reminders.

projects where it is not cost effective or practical to connect it to a video display. Here's how to go without a monitor.

The Raspberry Pi is often touted as an inexpensive computer, but if you don't have a monitor and other assorted peripherals already available, the cost soars quickly. Also, since one of the most appealing features of the Raspberry Pi for creative projects is its diminutive size, you're likely to discover that you need to run in "headless" mode: no monitor, no keyboard, and no mouse. Just a Pi flying solo (perhaps literally if you're building [Hack #44]!). That's when it's time to run headless.

CHANGE YOUR ROOT PASSWORD

Don't forget to change your root password early on. It's a good practice in general, but it's particularly important in headless mode. Most Raspberry Pi distros have well-known default root passwords.

In general, your eventually headless Pi projects will begin life connected to a monitor and input devices just to get everything ready. If nothing else, it seems like the easiest way to get the IP address, which is the first step to being able to SSH to the Raspberry Pi. However, if you use Pidora, you can go headless from the beginning, thanks to a configuration option that bypasses the first boot process and is meant specifically for going headless.

Once you've installed Pidora on your SD card (you can download the latest version from http://www.pidora.ca), create a file called `headless` in the partition named `boot`.

For a static IP address, list it along with the netmask and gateway in the `headless` file:

```
IPADDR=192.168.1.123
NETMASK=255.255.255.0
GATEWAY=192.168.1.1
```

```
RESIZE
```

If you would like to set the swap amount, add it here as well:

```
SWAP=512
```

If your Pi should obtain its IP address dynamically (DHCP), `headless` should stay empty. But then how do you find out what the IP address is? This is where Pidora's headless mode comes through for you!

Once you boot the Raspberry Pi with this headless file, the IP address will first flash through the speakers two minutes after powering on. Thirty seconds later, it will flash the IP address through the green OK/ACT LED. These functions are provided through `ip-info`, a package that contains the aptly named `ip-read` and `ip-flash`. The flashes indicate numbers in the following way:

- Digits 1–9 are indicated with short flashes (e.g., three short flashes is a 3).
- 0 is indicated with 10 short flashes.
- Digits are separated by a pause.
- Dot (.) is indicated with a long flash.

You can read more about the `ip-info` package and download it at https://github.com/ctyler/ip-info/.

As mentioned earlier, Pidora would usually run through the first boot process and have you set up a root password and another user. But that script will run only if input devices are found. Otherwise, the system configures the ethernet interface via IPv4 DHCP and assumes you'll set up any other preferences you would have made at first boot on your own.

from your project or just too lazy to walk across the room, you'll need to know how to SSH to your Raspberry Pi.

OpenSSH, the open source set of tools for secure communcation created by the OpenBSD project, is likely available in any distro you choose.

> If you're going a little retro, note that "Squeeze," the version of Raspbian before "Wheezy," didn't have SSH running by default.

If you aren't certain, all you have to do is attempt to SSH to your Pi, and you'll find out pretty quickly. Attach a monitor and keyboard, and then run:

```
$ service sshd status

'Redirecting to /bin/systemctl status  sshd.service
sshd.service - OpenSSH server daemon
      Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled)
          Active: active (running) since Wed 2013-02-13 13:06:40 EST; 28mi
         Process: 273 ExecStartPre=/usr/sbin/sshd-keygen (code=exited, statu
        Main PID: 280 (sshd)
          CGroup: name=systemd:/system/sshd.service
                  └─280 /usr/sbin/sshd -D
```

If your output doesn't look similar to that, it's quick to install. Here's the command on Fedora:

```
$ su -c 'yum install openssh-server openssh-clients'
```

And here's how to install it on Debian/Ubuntu:

```
$ su -c 'apt-get install ssh'
```

```
$ su -c 'chkconfig sshd on'
```

If you're not going headless from square one with the Pi, you can connect it to a monitor and run `ifconfig`. That's the simple way, assuming you've got a monitor and keyboard handy. Note that if you're using a newer version of Fedora or Pidora, you'll need to use `ip addr` instead.

Or check your router's default IP address, which is probably on a sticker somewhere on it or on a website if you search for your router brand. (192.168.0.1 is a common one.) You can also run `route -n` to find it. The numbers under Gateway on the line flagged UG are the default IP. Go to that address in a web browser, and you'll almost certainly find some sort of router control panel where you can see connected devices, including your Pi.

You could also use `nmap`, the network mapper tool. This is a fun way to learn a new tool if you haven't used it. That said, you should do your `nmap` learning only on your home network and not at the office, in the coffee shop, or anywhere else you're not in charge of said network. When you run `su -c nmap 192.168.1.1/24`, replacing the IP address with that of your network, you'll see a list of everything connected to that network. One of them will have a MAC address labeled Raspberry Pi Foundation, and it will list your Pi's IP address as well.

> If you're going to frequently connect via SSH, you'll want to simplify things by giving your Pi a static IP address (see [Hack #13]).

And now you're ready to connect to your Pi by running `ssh username@host`, where `username` is an account you've set up on the Raspberry Pi and `host` is the IP address you found or configured. If you haven't yet set up a user, you might need to refer to the default login. On Pidora, it's `root/raspberrypi`. On Raspbian-based systems, it's `pi/raspberry`.

```
The authenticity of host '192.168.1.174 (192.168.1.174)' can't be established
RSA key fingerprint is 78:75:1d:1c:a1:79:11:18:15:e5:04:08:15:16:23:42.
Are you sure you want to continue connecting (yes/no)?
```

It sounds a little ominous, but "yes" is the right answer, despite the "warning" that follows.

Now you're ready to use the command line to transfer files to and from your Raspberry Pi and to work on it almost as if you were working directly on it. If you'd like to be able to launch GUI interfaces over SSH, use `-X` when you connect:

```
$ ssh -X ruth@192.168.1.118
```

Most (but not all) graphical applications will work with this method, known as "X forwarding."

Hack 13. Give Your Pi a Static IP Address

If you always want to be able to connect to your Pi through the same IP address without looking it up, you'll need to assign it a static IP address (as opposed to a dynamically assigned one).

Many ISPs use *dynamic IP addressing*, which means that you get a different IP address each time you connect to the Internet. If you're connecting to the Pi over SSH regularly (see [Hack #12]), using VoIP (see [Hack #32]), or have other reasons to always have the same IP address, you'll want to set up static IP addressing.

In Pidora, you can either follow the instructions in [Hack #11] if you're running headless, or if you're not, edit the files in `/etc/sysconfig/network-scripts`.

You'll see the available network interfaces configurations listed as `ifcfg-<interface-name>`. Choose the one you'll be using for the connection and edit it in your favorite text editor, for example:

```
$ vi ifcfg-eth0
```

You'll see something like this:

```
DEVICE=eth0
BOOTPROTO=dhcp
ONBOOT=yes
NM_CONTROLLED=yes
```

You need to change the `BOOTPROTO` line from `dhcp` to `static`. Also make sure `ONBOOT` is set to `yes`. Then add `IPADDR`, `NETMASK`, `BROADCAST`, and `NETWORK` information like you would have in the headless file. Remember not to choose an IP address already in use elsewhere in your network. `NETMASK` is always `255.255.255.0`. `GATEWAY` is your router's IP address:

```
IPADDR=192.168.1.123
NETMASK=255.255.255.0
BROADCAST=192.168.1.255
GATEWAY=192.168.1.1
```

Finally, restart the network service to apply your new settings:

```
$ systemctl restart network.service
```

If you're using a Raspbian-based distro, you'll follow similar steps, just in a different place. Rather than looking for separate files, open `/etc/network/interfaces` (as root):

Then look for the line:

```
iface eth0 inet dhcp
```

Change `dhcp` to `static`, and add your static IP address, gateway, broadcast, and netmask:

```
iface eth0 inet static
address 192.168.1.123
gateway 192.168.1.1
broadcast 192.168.1.1
netmask 255.255.255.0
network 192.168.1.0
```

If you need a little help gathering these, you can find the current IP address, netmask, and broadcast by running `ifconfig` and noting the `inet addr`, `mask`, and `bcast`, respectively, while `route -n` will give you the gateway and network, which it calls `Destination`. (Again, on newer Fedora and Pidora versions, use `ip addr` instead of `ifconfig`.)

CHOOSING A UNIQUE STATIC IP ADDRESS

You should be sure to pick an IP address that is not already in use by any other devices on your network. Otherwise, your network connection will not work properly. When DHCP is in use, duplication is prevented, but it is possible to accidentally do it when setting the IP statically.

Additionally, you will need to manually specify a DNS server when setting a static IP address. DHCP configurations usually configure the DNS server for you, but there is no way for a static IP configuration to know what the DNS server is. To set the DNS server, edit `/etc/resolv.conf` (as root), and add the following line:

```
nameserver 11.23.58.13
```

After saving your changes, restart networking for the new settings to take effect:

```
$ su -c '/etc/init.d/networking restart'
```

You now have a static IP address that won't change each time you access the Internet.

Hack 14. Learn to Speak GPIO

GPIO stands for General-Purpose Input/Output, and its presence on your Raspberry Pi makes many hacks in this book possible. This hack helps dymystify it.

The Raspberry Pi contains standard connectors that you are probably familiar with (Ethernet, HDMI, audio, and USB), but it also includes 26 pins (in two rows of 13) that are intended to connect directly to lower level devices. These pins are called the GPIO (general-purpose input/output) pins, because they are programmable input/output pins intended for a wide range of purposes.

Practically, this means we can use the GPIO pins to connect almost *anything* to a Raspberry Pi. The header of these pins is labeled on the Raspberry Pi as P1, as shown in Figure 1-5.

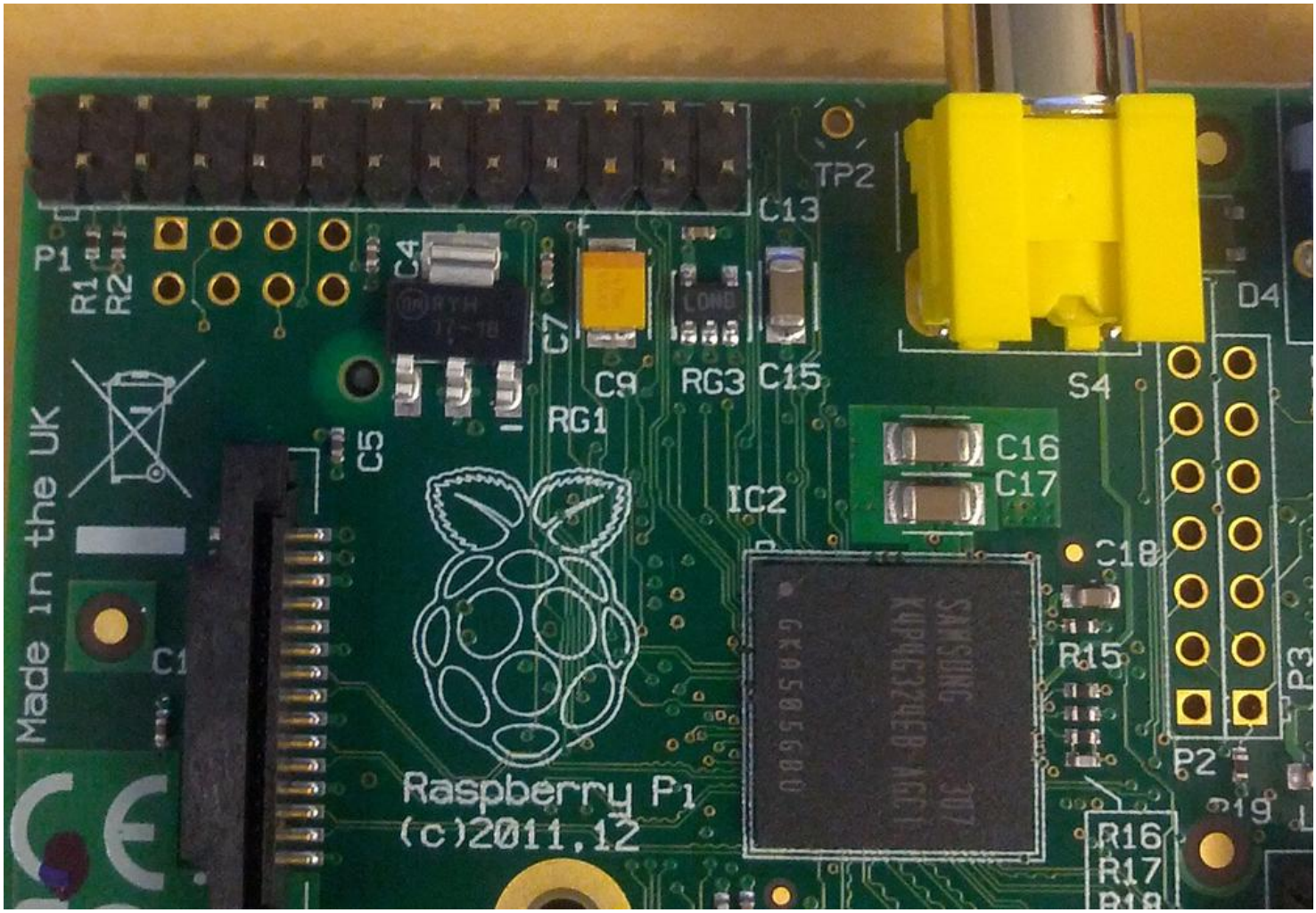

Figure 1-5. Raspberry Pi Model B with the GPIO header in the upper-left corner

Simple enough, right? Well, here's where it gets a little more confusing. There are two ways of numbering the GPIO pins on the Raspberry Pi.

Pin-Number Labeling

The first way to label the GPIO pins is to refer to the pin numbers on the P1 header on the Raspberry Pi board. If you look at the pins in Figure 1-5, Pin 1 is the first pin to the left in the bottom row, Pin 2 is the first pin to the left in the bottom row, and they continue to alternate in values to the right. In table form, the board pin numbers look like Table 1-5.

2	4	6	8	10	12	14	16	18	20	22	24	26
1	3	5	7	9	11	13	15	17	19	21	23	25

This method of labeling the Raspberry Pi GPIO pins by their board numbers is simple to understand by looking at the hardware. If you use this numbering scheme, you should prefix the pin numbers with the board label, "P1-".

BCM Labeling

But another labeling system for the Raspberry Pi GPIO pins uses the channel numbers on the Broadcom SOC. This system is referred to as the *BCM system*. It is the most common system in use, especially because the GPIO pins are not completely interchangeable and they are wired differently between the various models and revisions of the Raspberry Pi.

Table 1-6 shows the mappings of the BCM pin labels to the pins as shown in Figure 1-5 (for the Raspberry Pi Model B Revision 2, the current revision as of this writing).

Table 1-6. BCM pin labels (Raspberry Pi Model B revision 2)

5 V	5 V	GND	14 (TXD)	15 (RXD)	18	GND	23	24	GND	25	8	7
3.3 V	2 (SDA)	3 (SCL)	4	GND	17	27	22	3.3V	10 (MOSI)	9 (MISO)	11 (SCKL)	GND

Let's dig a little deeper. As you can see from the labeling, some of the pins are preconfigured for special purposes. The pins marked as 3.3 V and 5 V are power pins, with the voltage as labeled. The GND pins provide ground for wiring up circuits. BCM Pins 2 (P1-03) and 3 (P1-05) are pre-setup to provide I2C bus 1. BCM Pins 14

For these reasons (and also because it is the most common Raspberry Pi GPIO labeling scheme used on the Internet), this book uses the BCM pin labels.

We realize this might be confusing when you are wiring up devices to your Pi, but see the next section for a clever way to help you remember what is what.

Label Your Own GPIO Pins

GPIO should be simple, but the common labeling scheme (BCM) is so confusing and easy to forget. Here's a simple hack to make sure you always remember which pin goes where.

Dr. Simon Monk had a problem: he wanted to wire all sorts of temporary connections to his Raspberry Pi GPIO pins, but every time he wanted to do so, he had to go online and look up the BCM pin labels. Then there was the task of counting down the pins to find the right one, and while this sounds easy, trust us, you'll likely get this wrong just as he did.

To solve this problem, he created something called the Raspberry Leaf (shown in Figure 1-6). The Raspberry Leaf is a perfectly sized and scaled diagram of the Raspberry Pi GPIO pins, with the BCM labels next to them.

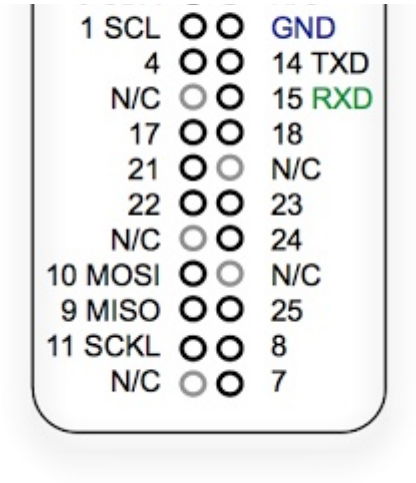

Figure 1-6. Raspberry Leaf, created by Simon Monk

You can photocopy and use this image for reference, but it's probably easier to download from this book's Git repository or the original PDF from Dr. Monk's website.

Hack 15. Connect GPIO Pins to a Breadboard

A solderless breadboard is a helpful friend when building electronics hacks, especially when you are prototyping or just testing out a device. Let's hack a simple connector to our Raspberry Pi.

While you can simply connect your Raspberry Pi GPIO pins to devices via common jumper wires, or solder wires directly between your add-on device and the GPIO pins, it is almost always helpful to have a little more space to work. Enter our old reliable friend, the solderless breadboard, shown in Figure 1-7. Even if you've never done an electronics project before, you may have seen this fellow with rows and columns of little holes in a rectangle of white plastic.

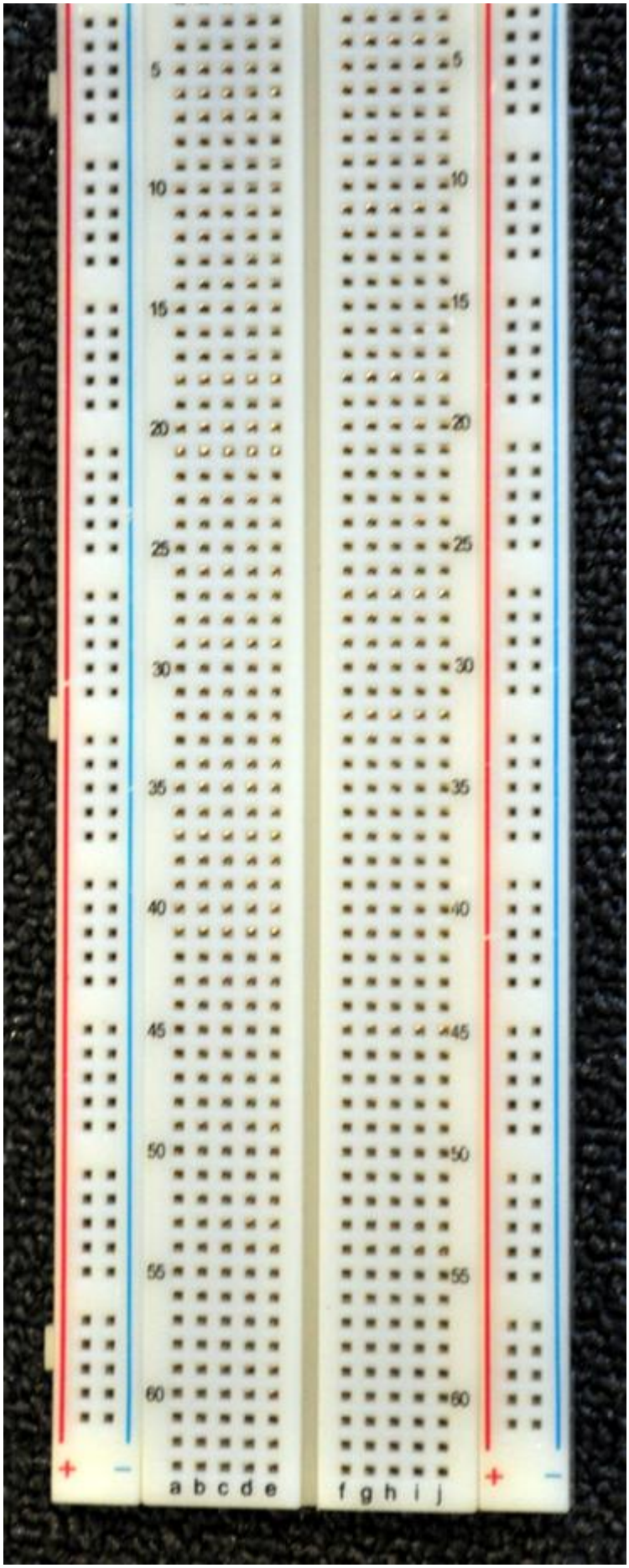

Figure 1-7. A breadboard

more holes? Just jump a wire from one row to another.

Additionally, most breadboards have vertical "rails" down each side, marked with red and black. These rails are intended to be used for power and ground connections, to simplify wiring circuits.

Our friends at Adafruit built a handy kit called the Pi Cobbler, which allows you to connect a standard 26 pin ribbon cable (just like you'd use on a PC motherboard) to a labeled printed circuit board (PCB) with a cable connector and individual pin breakouts. That PCB breakout board has pins that allow it to push right into your breadboard. Then, connect the cable to the Raspberry Pi GPIO pins and to the Cobbler PCB breakout board, and you can start connecting devices directly through your breadboard.

> The instructions in this hack are also inspired by Limor "Ladyada" Fried's excellent assembly tutorial.

Adafruit sells the Pi Cobbler in a couple variants:

Compact version
: Comes preassembled, so you can use it as soon as it arrives in the mail. (They used to sell this as an unassembled kit, but it doesn't look like they do anymore.)

T-Cobbler version
: Slightly larger, but covers less of the pins on the breadboard due to its "T" layout. At the time of this writing, the T-Cobbler is sold only as part of a kit (not preassembled).

If you end up with an unassembled kit for either of these versions, do not fret. It is easy to assemble it yourself. Here's everything you need:

- A breadboard
- Your Pi Cobbler kit, containing a blue PCB, a 26-pin ribbon cable, a black connector, and some male header pins (this might be in a single stick or in two smaller sticks)

If your male header pins (these are the metal pins with black plastic header in the middle, splitting the pins into one short and one long end) are in a single long stick, gently break off two pieces of 13 pins each. You can do this with your fingers or pliers. These correspond to the two pairs of 13 holes on the long edges of the Pi Cobbler PCB.

Also, go ahead and plug in your soldering iron and set it on a stand (see Soldering Reminders if you need some help or if it's been a while). Give it 5–10 minutes to come up to full temperature. If you have a fancy soldering iron with a temperature setting, Adafruit recommends you set it to 700 degrees Fahrenheit.

Place the Pi Cobbler PCB in front of you so that the pin labels (e.g., GND) are legible and oriented normally. On the T-Cobbler kit, the board is aligned like a T. On the original kit, the board's longer sides should be parallel to you.

The PCB is labeled with a box, indicating where the black header connector should be placed. Gently press the header into the box, making sure to align the notch in the header with the notch indicated in the box. On the original Cobbler PCB, the notch must be right next to the "21/27" label; on the T-Cobbler PCB, the notch must be between the two large round holes at the T junction point. You need to get this right, because if you get the notch backward, this will cause the pins to be reversed when the cable is connected between the Cobbler and the GPIO pins, and the labels on the Cobbler will all be wrong.

Keep a few things in mind:

- Solder, when heated, releases mildly toxic smoke. You should always solder in a well-ventilated area.
- The soldering iron gets hot. Very hot. (The actual temperature varies on the type, quality, and condition of your soldering iron.) It will burn you if you touch the tip of the iron, so always hold it by the handle, and don't lean in too close.
- Always assume a plugged-in soldering iron is hot and treat it that way. Heck, we usually assume unplugged soldering irons are hot, too.
- Every soldering iron should come with a stand. Use it. This keeps you from accidentally burning holes in your workstation, project, hand, leg...
- Keep a small, moist sponge handy, and as needed, use it to wipe off excess solder from the tip of the hot iron. If the tip of the iron becomes coated in solder, it will no longer work effectively.
- You don't need to put a giant blob of solder down to make a good connection. When heated, solder quickly becomes a liquid and will flow into heated connections. Just a tiny bit will do. Practice will help you realize how much to apply.

Flip over the PCB, with header connector still in place, so that it is now sitting on the header. You should see little bits of the 26 connector pins poking out from 26 metal rings on the PCB. Press the tip of your soldering iron simultaneously against a pair of the rings and pins. Hold it there for a few seconds to heat up the metal, and then touch some solder against the tip of the iron. The solder will melt instantly, liquify, and flow between the pin and the ring, making a complete connection.

You want to use enough solder so that you cannot see air between the pin and the ring, but not so much that you make a connection between neighboring pins. Really, it doesn't take much, just a tiny bit. This solder will be completing the electrical connection, but it will also be providing a mechanical bond that holds the device together. Repeat this process for all 26 pins, until the header connector is neatly soldered to the PCB, and then put your soldering iron back on its stand (you'll use it again in a moment).

You want to do this so that they are in the same spacing and alignment as they appear on the PCB Cobbler. For the original Cobbler, this is about five breadboard rows apart; for the T-Cobbler, this is only three rows apart.

Flip the PCB back over and set it into the short ends of the male header pins. The breadboard is acting as a stand for us now. Push the PCB gently down until all of the pins are poking through the labeled rings, and the PCB is resting up against the plastic header middles. Pick your soldering iron up again, and solder each of these 26 rings and pins.

When you're finished, clean off the tip of your soldering iron with a moist sponge and unplug it. Put it back on the stand to cool off. You can now connect the ribbon cable between the completed Pi Cobbler and the Raspberry Pi GPIO pins. You'll notice that the cable will only go into the Pi Cobbler one way, because of the notch on the connector. However, be careful, because the Raspberry Pi GPIO pins do not have any connector, and the cable can connect two possible ways. The ribbon cable included in your kit will have two indicators to help you align it properly:

- The ribbon cable has one wire of a different color. This uniquely colored wire should be on the edge closest to the SD card slot on the Raspberry Pi.
- Both ends of the ribbon cable have a notched connector. The notch on the connector going to the Pi should be pointed *toward* the Raspberry Pi logo on the board, as shown in Figure 1-8, never away from it.

The finished and connected Pi Cobbler will look something like Figure 1-9 (this is an original Pi Cobbler).

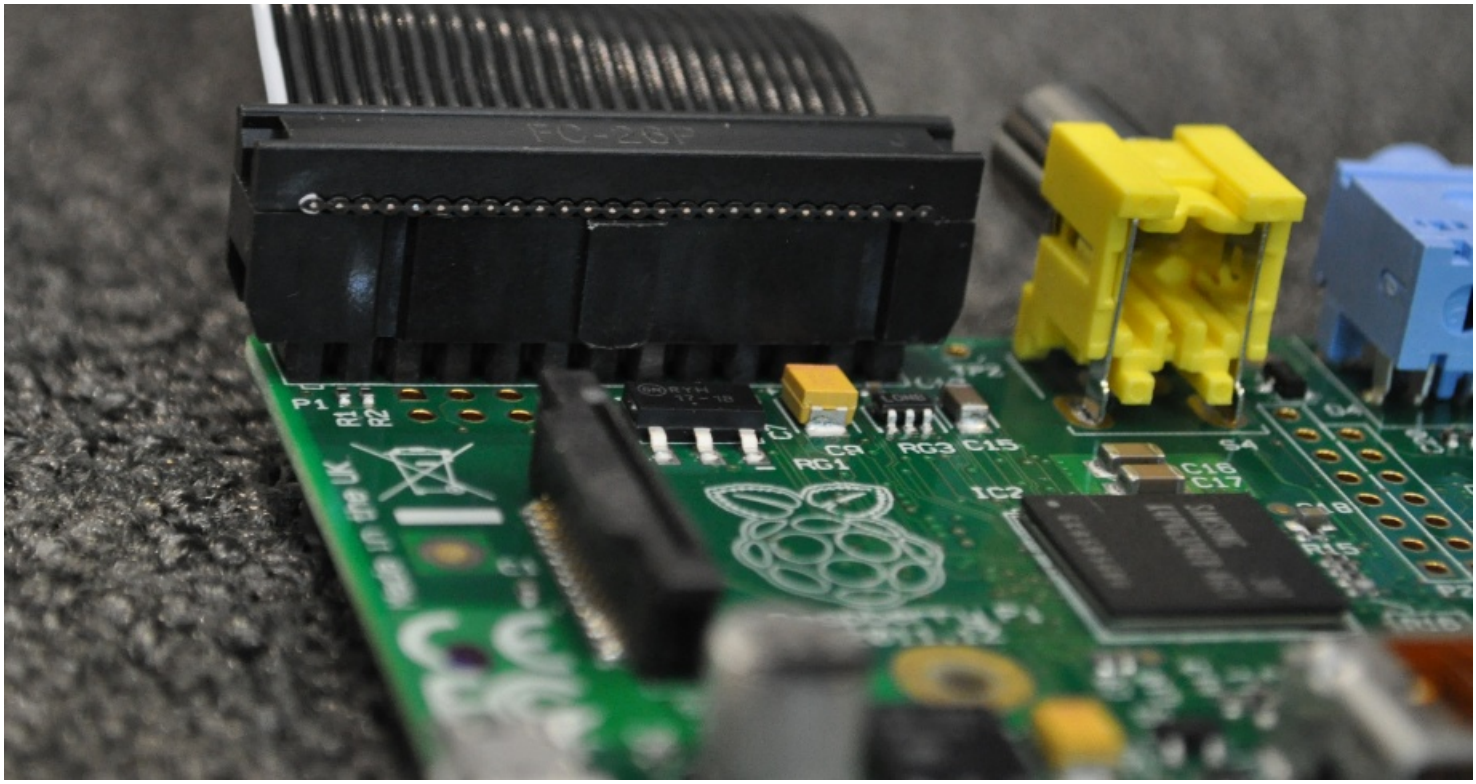

Figure 1-8. Close-up of a properly connected Cobbler ribbon cable

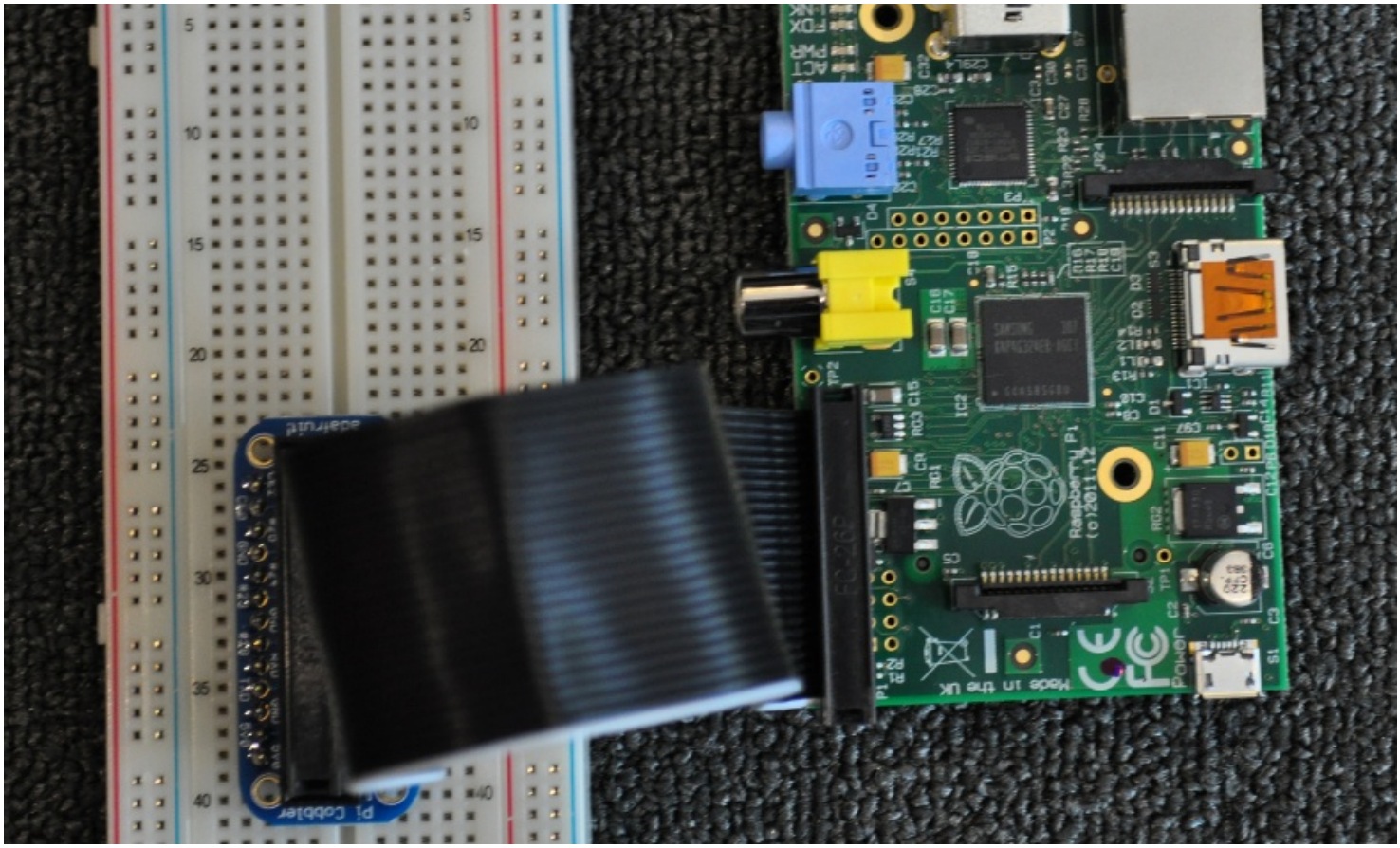

Figure 1-9. Completed and connected Pi Cobbler

It might not seem like much, but trust us, when you are wiring up multiple devices to the Raspberry Pi GPIO pins, being able to easily use a breadboard (and see the GPIO labels at a glance) will make you happy that you completed this hack.

GPIO Quick Reference

If you turn your Pi so that the GPIO pins are in the upper right, the pins are numbered from top to bottom, with odd numbers on the left and even on the right (thus, the first row is 1 and 2, second row is 3 and 4, etc.). Note that these do not correspond to GPIO numbers; for example, GPIO 22 is on pin 15. Table 1-7 explains the purposes of the pins of a Version 2 board.

1	3.3 V power
2	5 V power
3	GPIO 2 (SDA)
4	5 V power
5	GPIO 3 (SCL)
6	Ground
7	GPIO 4 (GPCLK0)
8	GPIO 14 (TXD)
9	Ground
10	GPIO 15 (RXD)
11	GPIO 17
12	GPIO 18 (PCM_CLK)
13	GPIO 27

15	GPIO 22
16	GPIO 23
17	3.3 V power
18	GPIO 24
19	GPIO 10 (MOSI)
20	Ground
21	GPIO 9 (MISO)
22	GPIO 25
23	GPIO 11 (SCLK)
24	GPIO 8 (CE0)
25	Ground
26	GPIO 7 (CE1)

Inter-Integrated Circuit (I2C)

The I2C interface (SDA and SCL), which you can access through pins 3 and 5, is a connection for low-speed peripherals or sensors. You can have multiple devices connected through the same pins.

Pulse-Width Modulation (PWM)

Pin 12 offers control for motors similar to analog control through pulse-width modulation (labeled PCM_CLK). For some purposes, you can achieve the same effect through software, which may be useful since the Pi has only one PWM pin.

Universal Asynchronous Receiver/Transmitter (UART)

The UART pins (14/TXD and 15/RXD) are used for serial console access. If you don't need that, you can switch them to GPIO for an extra two GPIO pins. This is also true of the I2C and SPI pins, but you're least likely to want to use the UART pins.

Serial Peripheral Interface Bus (SPI)

The SPI pins are the pins you'll use for some types of sensors or attaching other devices. SPI operates in master/slave fashion:

- 19 - Master Out, Slave In (MOSI)
- 21 - Master In, Slave Out (MISO)
- 23 - Serial Clock (SCLK)
- 24 - CE0 (chip select)
- 26 - CE1 (chip select)

The two chip select pins mean you can control two SPI devices.

miliar with Arduino wiring.

Hack 16. Add a USB Serial Console

Arguably the most common way to access embedded devices like the Raspberry Pi is via the built-in serial device. This easy hack gives you a USB serial console from your Raspberry Pi.

Almost all of the common embedded computers and microcontrollers available today have built-in Universally Asynchrononous Receiver/Transmitters (UARTs). The UART provides a mechanism for receiving and transmitting serial data, one bit at a time. This method of serial communication is sometimes referred to as transistor-transistor logic (TTL) serial. The data rate varies by device, but it is measured in bits per second. The Raspberry Pi has a built-in UART connected to BCM Pins 14 (TXD) and 15 (RXD), with a data rate of 115200bps (or baud).

HEY, THAT'S NOT REALLY BAUD!

OK, from a semantic point of view, yes, baud is the unit of symbol rate, which is not always identical to gross bit rate (bps). Wikipedia says that baud is "the number of distinct symbol changes (signaling events) made to the transmission medium per second in a digitally modulated signal or a line code." Then it goes on for several more pages of mathematical distinction about baud, which may be fascinating to you or may put you to sleep.

The key takeaway is this: the computer and electronics industry has been widely misusing the term baud for about 40 years now. As a result, lots of software and hardware uses bps and baud interchangeably. In the specific case of the USB serial console on the Raspberry Pi, the data rate in bps and the baud rate are the same. The units here don't matter to us as much as making sure you have a functional serial console, and for that, you need to know that the data rate is 115200. Bits, baud, giant hamsters of doom, pick your units as you will, but 115200 is the magic number for the Raspberry Pi UART serial device.

If you've been using computers for a few years, you probably remember when almost every computer came with an RS-232 serial port, but in the last few years, these ports have been disappearing, and most laptops no longer include them (or they only have them on the optional laptop dock). Believe it or not, for connecting to the Raspberry Pi UART serial port, this is actually a good thing. The Broadcom chip

signal levels before it would work.

RS-232 OR BUST!

Here's a pretty good tutorial on how to build a 3.3 V to RS-232 level converter.

The good news is that there is a better way to connect the Raspberry Pi UART serial port to your computer: USB! Adafruit sells a wonderful USB-to-TTL Serial Cable, which connects directly to the GPIO pins on the Raspberry Pi and provides a USB serial device on the other end. This cable has four female jumper connectors on one end (the end that doesn't have a USB connector). These jumpers have color-coded wires: red for 5 V power, black for ground (GND), green for receiving data into the Raspberry Pi (RXD), and white for transmitting data from the Raspberry Pi (TXD). You might also notice that the USB connector end is larger than normal, because it also has a USB-to-Serial conversion chip inside it.

To make the physical connection, you simply need to connect three of the female jumper connectors directly to the appropriate pins on the Raspberry Pi GPIO. The white transmitting wire goes into the TXD port (BCM Pin 14 (P1-08)), and the green receiving wire goes into the RXD port (BCM Pin 15 (P1-10)). The black ground wire can go into any of the GND pins, but for simplicity, we recommend you put it in the GND pin immediately to the left of the TXD port (P1-06). You can confirm your wiring by comparing it to Figure 1-10.

will power itself with 5 V at 500 mA directly from the USB port. You may also note that we said that the Raspberry Pi UART uses 3.3 V logic levels, not 5 V, and this is true, but the receiving and transmitting wires are already converted to 3.3 V. It all works fine, stop noting and move on.

There is, however, a cool mini hack you can do with that red power wire. Because 5 V is going across that red wire, you can use it to power the Raspberry Pi, instead of doing so via the normal mini-B USB connector. Just connect the red wire to the 5 V pin (P1-04), then without any other power source connected, plug the USB-to-TTL Serial cable into your laptop. The Raspberry Pi will boot up!

This is really just a parlor trick, because the power coming off that red wire is not the ideal way to power the Raspberry Pi for a number of reasons, as discussed in Think Twice Before Using the GPIO to Power the Pi.

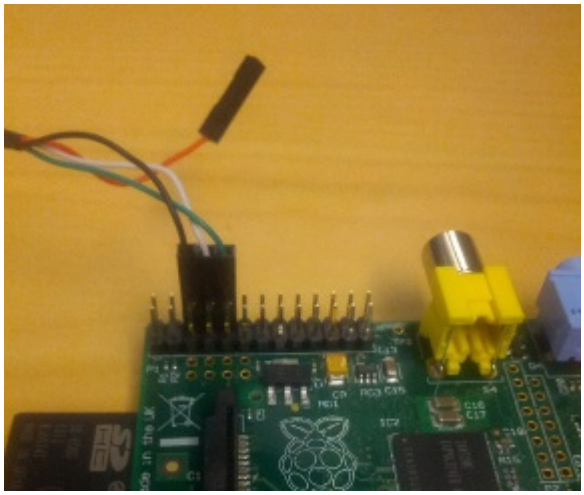

Figure 1-10. A properly wired USB to TTL serial cable

Now, go ahead and connect the USB connector to your computer.

To connect to the UART serial device, you first need to know its device name. The kernel assigns it a device name when the USB serial driver successfully loads (which it should have already done when you inserted the USB end of the cable), so you just need to look through the output from dmesg.

Specifically, we know that the device name will be `ttyUSB#`, where # is a number. It's probably `ttyUSB0`, but let's look to be sure. If you have multiple USB serial devices present on your system (you naughty super hacker, you), you're looking for the one with the pl2303 converter type. If you have more than one pl2303 converter

```
$ dmesg | grep -B2 ttyUSB
[23882.896558] usbserial: USB Serial support registered for pl2303
[23882.896578] pl2303 1-1.5.1:1.0: pl2303 converter detected
[23882.898285] usb 1-1.5.1: pl2303 converter now attached to ttyUSB0
```

Sure enough, our device is `ttyUSB0`. This means that the full device node name is `/dev/ttyUSB0`. Unprivileged users do not normally have access to `/dev/ttyUSB#` devices; you need to be in a special group. If you look at the file permissions on the device node name, you will see that it is owned by `root` and access is granted to users in the `dialout` group:

```
$ ls -l /dev/ttyUSB0
crw-rw----T 1 root dialout 188, 0 Aug 22 19:11 /dev/ttyUSB0
```

You can either connect to the `/dev/ttyUSB0` device using the root account (via `su` or `sudo`), or you can add your normal user to the `dialout` group. To add your user to the dialout group, run:

```
$ su -c 'usermod -a -G dialout $USER'
```

This will not take effect in your terminal sessions until they are restarted. Either log out and log in again, or reboot your Linux system.

can give you a few tips anyway.

Windows and Mac OS X systems will need to install PL2303HXA drivers, which you can download for Windows XP/Vista/7 and Mac OS X.

According to the vendor, Windows 8 is *not* supported for this device. Maybe if you're a Windows 8 user, this is a good time to consider dual booting to Linux?

As far as terminal software goes, if you are using Windows XP or older, it comes with a program called Hyperterminal that can connect to a serial console. If you are using a newer version of Windows, you'll need to download a third-party terminal program. We recommend PuTTY. For Mac OS X, you can either use screen in the same way that we've described for Linux, or you can try ZOC.

Now it's time to connect to the Raspberry Pi UART serial device. You'll need to use a client that supports a serial connection; there are lots and lots out there, but the two common ones are minicom and screen.

Minicom

Minicom was written to look like Telix, a popular MS-DOS terminal program that was probably written before you were born. We now feel old(er). It has that MS-DOS look and feel to it—namely, it is old, crufty, and confusing—but it does work. To install it on Fedora:

```
$ su -c 'yum install minicom -y'
```

or on Debian/Ubuntu:

```
$ su -c 'apt-get install minicom'
```

Once installed, to use minicom to connect to the Raspberry Pi UART serial device, run:

```
$ minicom -b 115200 -o -D /dev/ttyUSB0
```

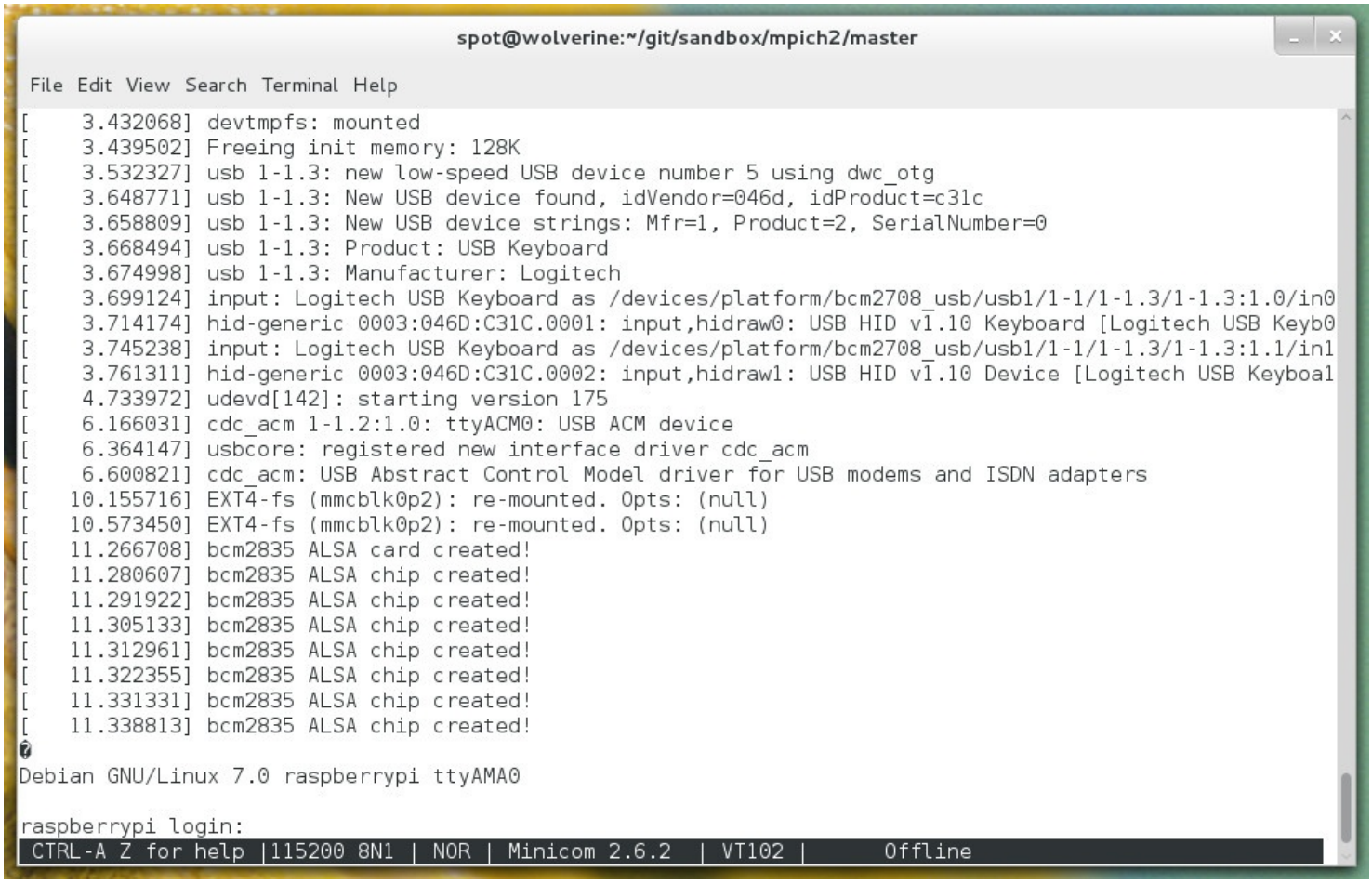

Figure 1-11. minicom interfacing with the Raspberry Pi UART serial device

You can exit minicom with Control-A X, or get into its somewhat helpful help menu with Control-A Z. If you end up manually configuring minicom, just leave the Parity/Bits at 8N1, and disable the Software Flow Control. You don't need to go in there though. There are bats in there.

Screen

Screen is a powerful utility, normally used to multiplex multiple virtual consoles. If this were *Linux in a Nutshell*, we'd have a whole chapter about using screen. Since this is a different book, we'll just focus on using it as a serial terminal client.

To install it on Fedora, run:

```
$ su -c 'yum install screen -y'
```

```
$ su -c 'apt-get install screen'
```

Once installed, to use screen to connect to the Raspberry Pi UART serial device, run:

```
$ screen /dev/ttyUSB0 115200
```

To exit the screen session, type Control-A K.

Using the Serial Device as a Login Console

Raspbian preconfigures the UART serial device as a login-capable console, but Pidora does not. To enable the UART serial device as a login console session, run:

```
$ su -c 'systemctl start serial-getty@ttyAMA0.service'
```

This will turn it on immediately. If you have a serial cable connected, you should see a login prompt appear. To make this login session permanent (and automatically loaded on boot), simply run:

```
$ su -c 'ln -snf /usr/lib/systemd/system/serial-getty@.service \
  /etc/systemd/system/getty.target.wants/serial-getty@ttyAMA0.service'
```

By making that symbolic link in the systemd directory tree, you are telling systemd to start the ttyAMA0 device as a “getty” or login console.

writer for input and a printer for output for use as telegraph machines, but as computers evolved, these devices found a use as a method of inputting data to a computer. The teletype was also widely used in a “receive only” format in newsrooms in the 1940s and 1950s. The “clickety clack” sound effect of news coming in “over the wire” is that of a teletype. Eventually, these devices became proper serial consoles, and they are the reason why we call console devices in Linux “TTYs”.

Both Pidora and Raspbian come with the UART serial device preconfigured as a console for kernel messages. You can see this in the `/boot/cmdline.txt` file:

```
dwc_otg.lpm_enable=0 console=ttyAMA0,115200 kgdboc=ttyAMA0,115200 console=tt
```

The `console` value tells the Linux kernel where to output messages at boot time, and the `kgdboc` value enables kernel debugging over that console.

If you are working on a project that wants dedicated access to the UART serial device, you will probably want to remove the `console=ttyAMA0,115200` and `kgdboc=ttyAMA0,115200` entries from `/boot/cmdline.txt` and reboot your Raspberry Pi. If you do not, you will get unexpected line noise across the serial line from the Linux kernel that your program/project is probably not ready to deal with.

If you have enabled the serial devices as a login console, you will also want to disable that. To disable it on Raspbian, comment out the following lines in `/etc/inittab` (by changing the line to start with a #):

```
T0:23:respawn:/sbin/getty -L ttyAMA0 115200 vt100
```

On Pidora, you simply need to remove the systemd getty service symlink for the `ttyAMA0` device, by running:

```
$ su -c 'rm -f /etc/systemd/system/getty.target.wants/serial-getty@ttyAMA0.servic
```

Weird Noise (or Missing Signal) on the Serial Connection?

Both Ubuntu and Fedora include a piece of software called `ModemManager`, which handles setting up all sorts of modem devices, from old dial-up modems to more modern 3G/4G devices. Unfortunately, quite a few current modems just have the same generic converter chip that our USB serial connector cable uses (the pl2303) stuck in front of the modem.

As a result, `ModemManager` will try to access it (and keep trying, and keep trying …), because it has no way of knowing that the device behind `/dev/ttyUSB0` is a modem, or a Braille terminal, or in our case, a Raspberry Pi. This might prevent you from being able to open `/dev/ttyUSB0`, or it might simply cause noise to appear across the console.

Since we know what that USB device is connecting to, we can tell `ModemManager` to leave it alone and explicitly blacklist our connector device with udev rules.

As root, edit the `/lib/udev/rules.d/77-mm-usb-device-blacklist.rules` file, and add these lines before the `LABEL="mm_usb_device_blacklist_end"` line at the bottom:

```
# Adafruit USB to TTL Serial Cable (PL2303HXA)
ATTRS{idVendor}=="067b", ATTRS{idProduct}=="2303", ENV{ID_MM_DEVICE_IGNORE}=
```

If you use a different USB serial cable from the Adafruit cable, and you're trying to fix this problem, you should be able to run the `lsusb` application (from the `usbutils` package) to determine the `idVendor` and `idProduct` string (they will show up in the output in the syntax `XXXX:YYYY` where `XXXX` is the `idVendor` and `YYYY` is the `idProduct`).

For example, the `lsusb` value for the Adafruit USB serial cable looks like this:

Udev automatically detects changes to rules files, so changes take effect immediately without requiring udev to be restarted. That said, sometimes you need to reboot the Linux system for udev to reread its rules. Either way, whenever udev reads in its new rules, the USB serial converter device should now be blacklisted, and ModemManager should ignore it from then on out.

Hack 17. Add a Reset Button

Perhaps you've noticed your Pi lacks something pretty common among electronics: a power switch. The Model B revision 2 boards come with a small fix.

It's somewhere between vaguely uncomfortable and outright inconvenient to remove the power supply from your computer as an on/off switch, but that's what you do on the Raspberry Pi. One easy fix, regardless of what board you have, is to plug it into a power strip with an on/off switch and use that. But with the Model B revision 2 boards, you have another option.

One of the added features on these boards is labeled P6. It's easy to miss. P6 is just two small holes on the opposite side of the board from the GPIO, near the HDMI port (see Figure 1-12).

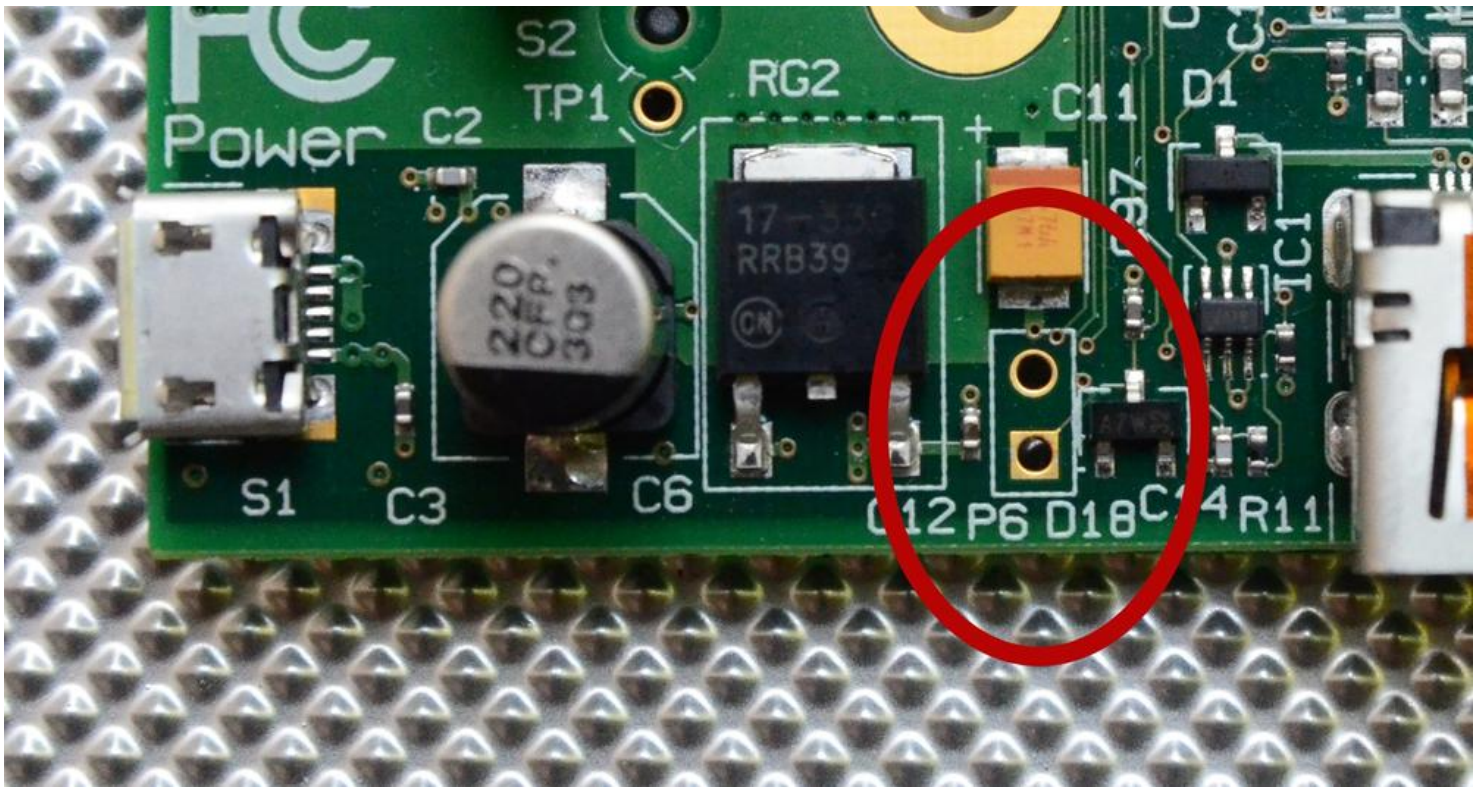

Figure 1-12. P6 holes

If you solder a pair of header pins into these two holes (see Soldering Reminders if you're new to soldering), you have a reset switch, as shown in Figure 1-13. Just use a metal object to connect the two pins and short them.

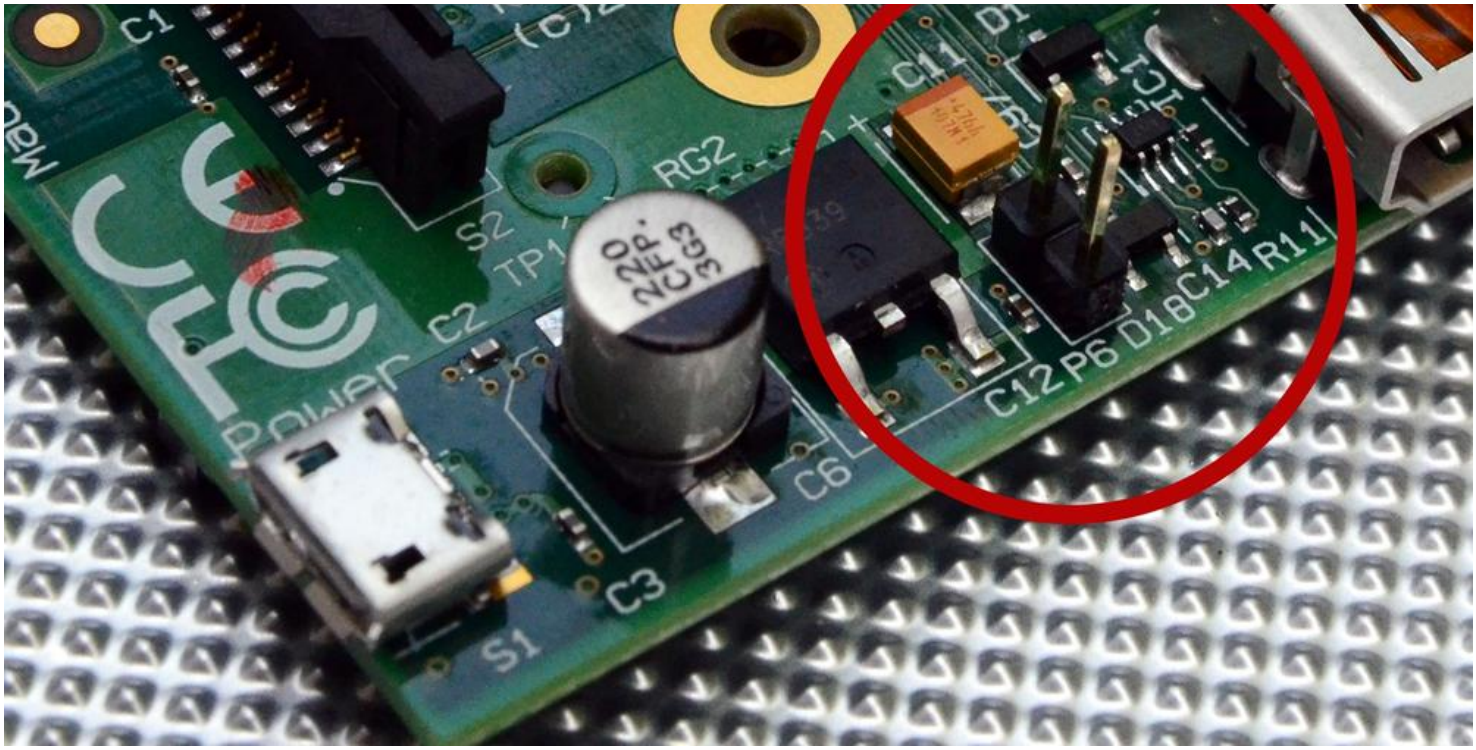

Figure 1-13. Reset pins in place

This short will also reset the CPU from shutdown, causing the Pi to start.

Hack 18. Get Power to the Pi on the Move

Power doesn't have to mean a plug in the wall. You have a few more options to increase portability.

To get power to your Pi, you need five stable volts at 700 mA through a Type B Micro USB plug. As mentioned elsewhere, it's not a bad idea to get a power adapter specifically intended for the Raspberry Pi, though your phone charger or other similar adapter will likely work.

mended but work out for some people. For example, powering your Pi through your laptop's USB port. It doesn't *officially* work and isn't recommended, but we've seen it work. Trying to push power through the GPIO isn't a great idea either. See Get Power Through GPIO Safely for more on power and the GPIO pins.

We've also heard of success with power over Ethernet and even intentionally backpowering through USB to power the Pi. Again, not the recommended method, but hacks are about trying something new. As long as you conduct your electrical experimentation safely, the worst you'll end up with is the loss of a $35 board but the gain of a good story, possibly featuring smoke, which enhances any story.

But if you want portability, what you need is a battery pack. You might think of these devices as emergency power for your cell phone. They come in assorted strengths, shapes, sizes, and colors, but in the end, they're two things: power and a USB port, which are the two things that you need to power the Pi.

Look for one that has 5V regulated output. We use the New Trent iCarrier (IMP120D), a 12,000 mAh pattery pack with two USB ports and an on/off button. Depending on activity on the Pi, that's enough power to last 14– 16 hours or more.

This portable power is critical to some of the hacks in this book, most notably [Hack #44]. It's an enhancement for others. And worst case, it's spare juice for your phone.

Hack 19. Test Your Might (in Volts)

If you aren't already friends with a multimeter, you will be soon. Pi projects all need power, and the Pi provides a way for you to check the voltage of the board on the board.

Assuming you don't have a few power supplies around to swap out, or if you're determined to get a particular and unusual power source working, you're going to want to check the voltage on the Raspberry Pi.

There are two test points on your Raspberry Pi for just such a need, as shown in Figure 1-14. TP1 is the 5 V point and TP2 is ground.

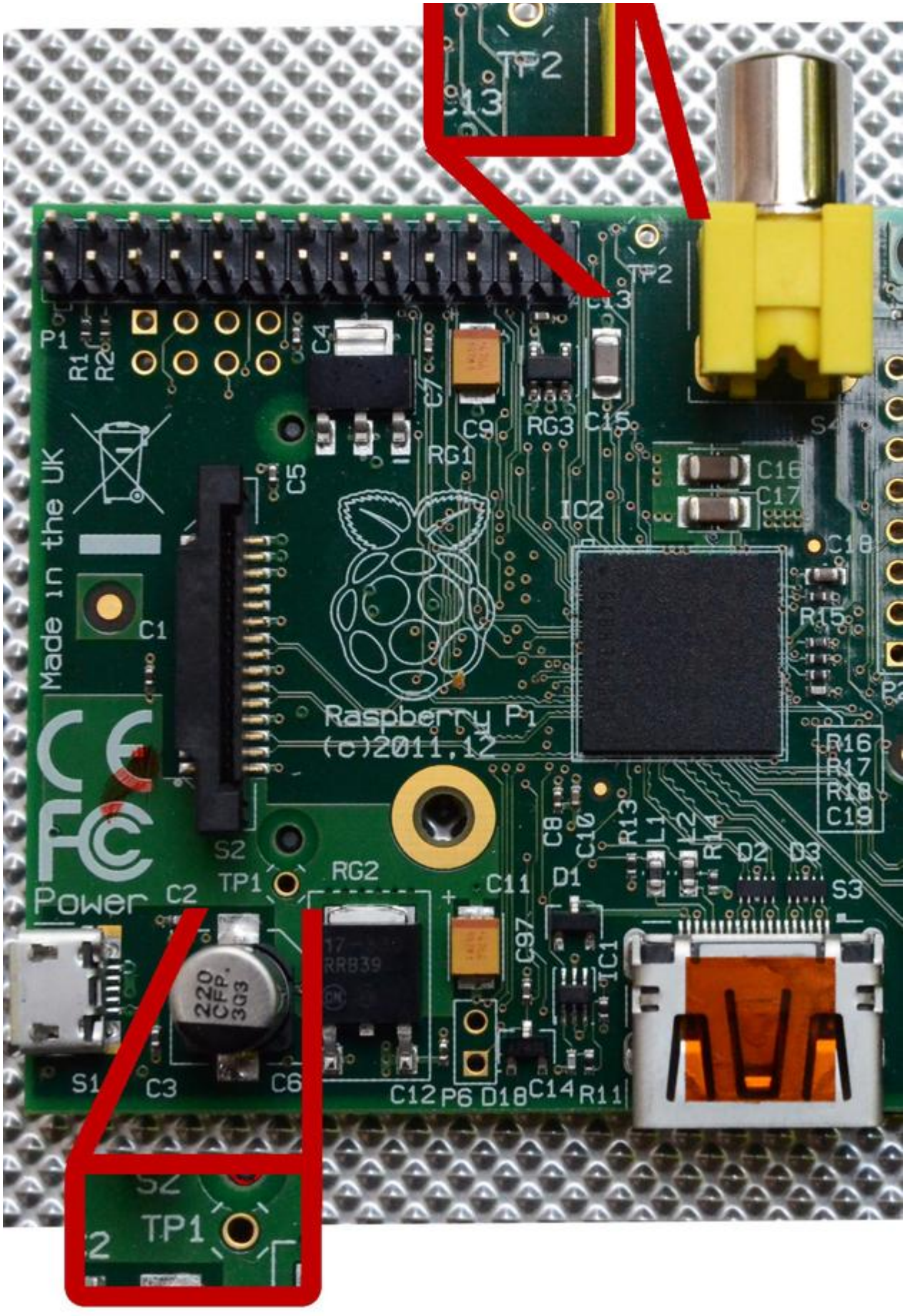

Figure 1-14. Test point locations (a.k.a. “electronics vision test”)

peripherals).

2. Set your multimeter range to 20 V.
3. Touch the red lead of your multimeter to TP1 and the black lead to TP2 (as shown in Figure 1-15).

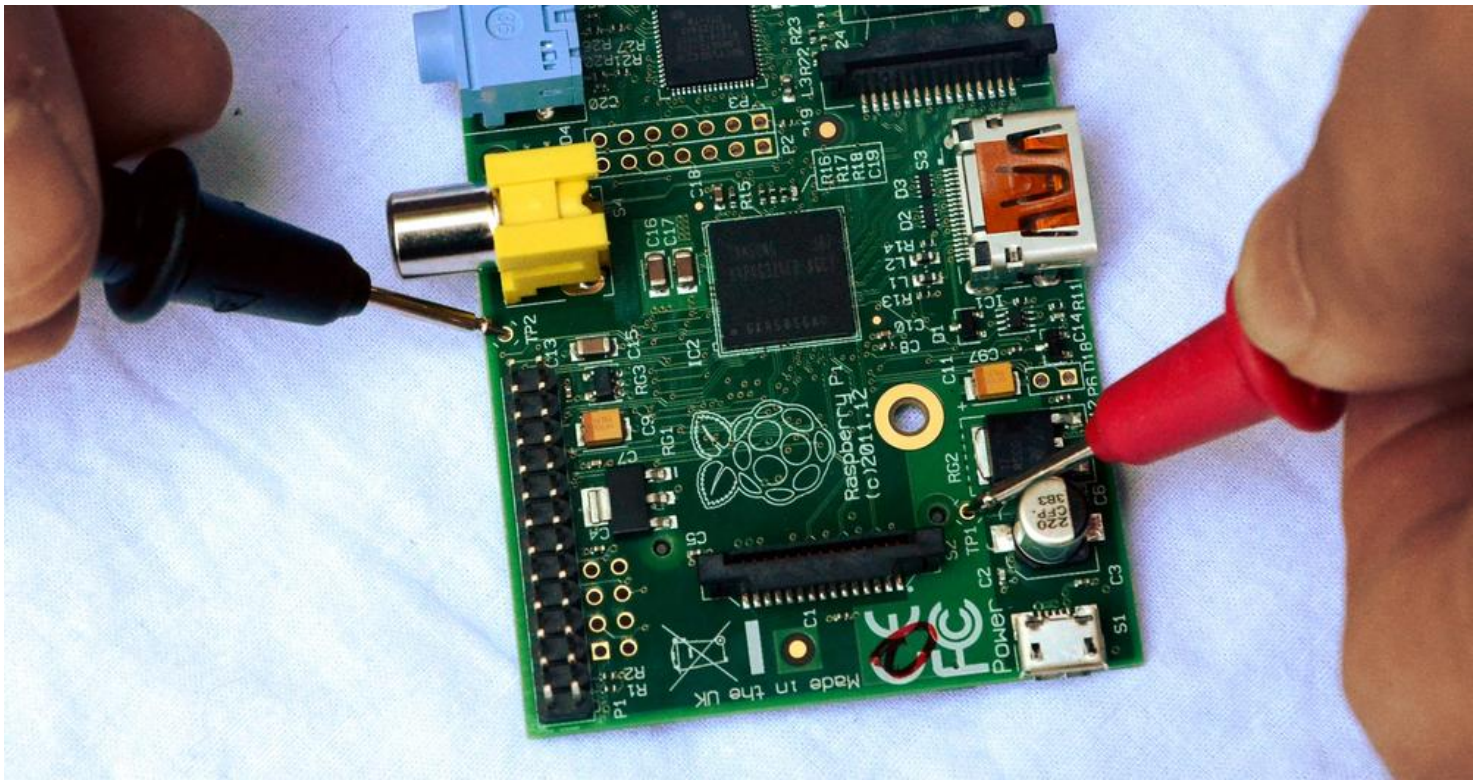

Figure 1-15. Testing the voltage

The Pi needs a good 5 V supply but has a tolerance of +/-0.25 V or so. That means that, at a minimum, you should be seeing 4.75 V, preferably more like 4.8 V or more. Below that, and either your peripherals will start acting up or the Pi might not even boot at all. It might also reboot spontaneously.

You can also try unplugging various peripherals, using different monitors, removing Ethernet, etc. Test again to see how the result changes.

your electronics. If that's the case, it's worth some light reading about how all this power stuff works. But at the most basic level, you'll want to understand these three terms and the relationship among them:

Voltage
: The difference in charge between two points

Current
: The rate at which charge is flowing

Resistance
: A material's tendency to resist the flow of charge (current)

If you know two of them, you can figure out the third, thanks to Ohm's law, which says that V=I*R, or the voltage (volts) is equal to the current (amps) times the resistance (ohms). Those who are mathematically disinclined can get an assist from the handy tool at the online Ohm's Law Calculator.

Keep this in mind when things act funny and you're not sure why: it might be worth a quick power check.

You can also use the Pi's test points to test its polyfuse. A *polyfuse* is a type of fuse that can repair itself after it has been blown. The Raspberry Pi has at least one of these, labeled F3 on the bottom of the board, as shown in Figure 1-16.

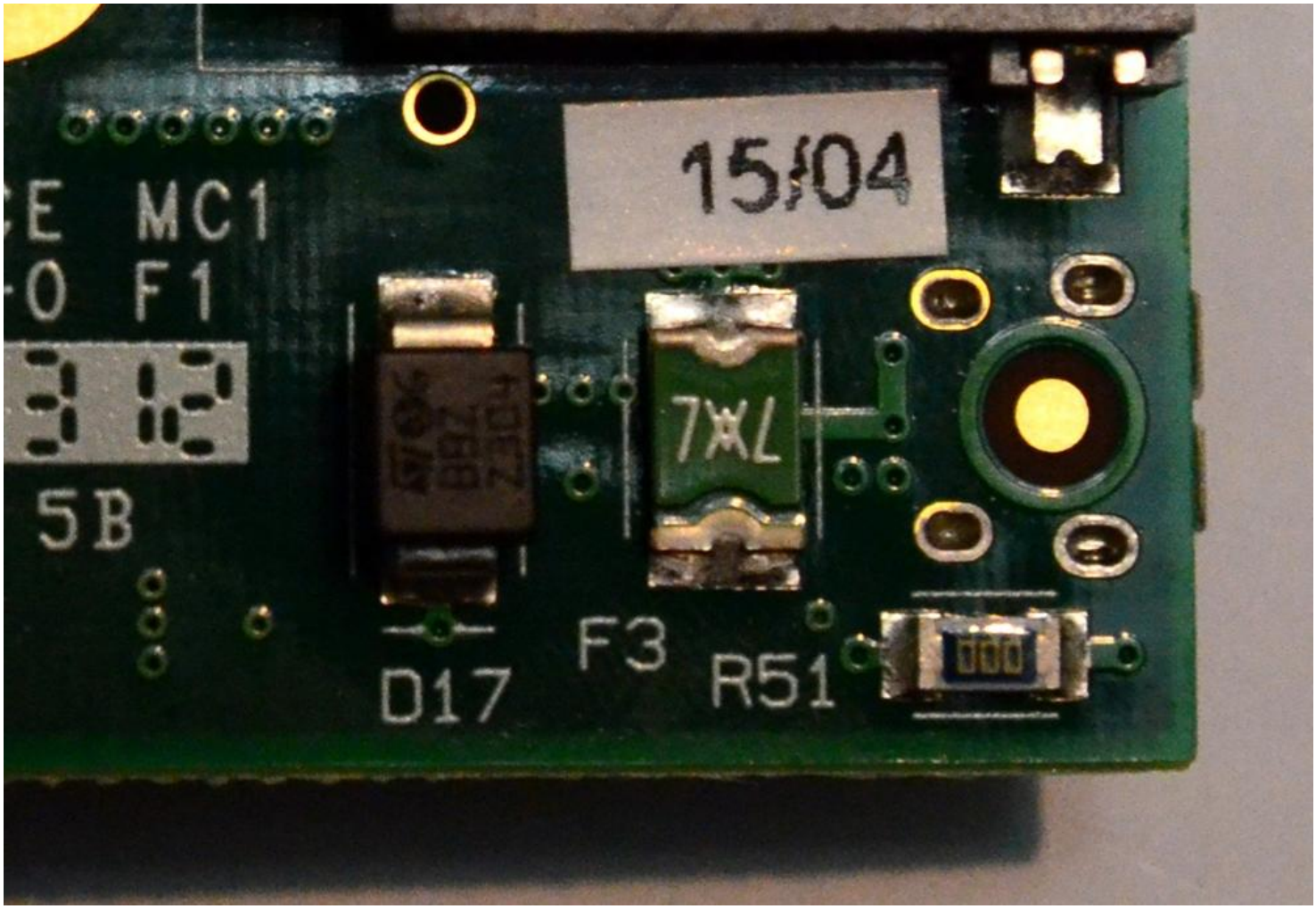

Figure 1-16. F3 polyfuse beneath SD card slot

Earlier Pis also had two on the USB ports, which have since been replaced by 0 Ohm resistors. They were known as F1 and F2.

You'll find F3 to the left of the SD card slot if you turn the Pi over and hold the SD slot toward you.

How long it takes for it to "heal" is variable, as much as a few days, and they can be permently damaged. They are replaceable, though. To test whether you're having a problem with the F3 polyfuse:

1. Remove the SD card and all of the peripherals, unlike testing the voltage, but leave it plugged in.

(since the board is upside down) and the other to the metallic part of F3 facing the SD card slot. This will tell you the voltage coming from the fuse.

4. Next, touch one lead to the back side of TP2 and the other to the metallic part of F3 facing the outside of the board. Check the voltage. This will tell you the voltage coming in.

It's normal for the reading on F3 to be 0.2 V lower than the power coming in, but any more than that indicates a problem with the polyfuse.

Hack 20. Add Additional Memory with Swap

Need a little more memory on your Raspberry Pi? Swap will let you trade disk space for memory.

Linux has long inclued the concept of *swap*, where the kernel is capable of moving memory pages between RAM and disk. In practical application, this provides more usable memory to the OS (at the cost of disk space). Because the Raspberry Pi Model B only has 512 MB of memory, the idea of adding swap files (or partitions) to increase the usable memory is compelling.

SWAP HAZARDS

Before we begin, you should be aware of a downside to this approach. Swap is only as fast as the speed of the storage device that it is written to. It also is a highly write-intensive operation. From a practical perspective, this means that if you add swap to your SD card in your Raspberry Pi, it will cause the overall disk performance to drop significantly and notably shorten the life of the SD card.

Because the entire OS on the Raspberry Pi runs off the SD card, we strongly recommend that you not place swap files or partitions on the SD card. These downsides also apply to a USB flash drive connected to the Raspberry Pi. Because they are also flash-based storage devices, adding swap files from USB flash drives will cause the same overall performance slowdowns and shortening of life span. If you really want to add swap, the best possible case is over an actual external hard drive connected via a SATA-to-USB converter.

swapfile, change the value from `100` to a larger value (depending on the free space on your SD card). Alternatively, you can disable this option by changingthe value to ++0.

Any changes to this value will not take effect until you run the following commands:

```
$ /etc/init.d/dphys-swapfile stop
$ /etc/init.d/dphys-swapfile start
```

Pidora configures 512 MB of swap by default at firstboot (unless the user specifies otherwise). This is placed in the file `/swap0` and configured in `/etc/fstab` by the rootfs-resize service.

For other Linux distributions (or to place a swapfile on a different location), you will need to manually create the swapfile:

```
$ sudo dd if=/dev/zero of=/path/to/swapfile bs=1M count=1024
$ sudo mkswap /path/to/swapfile
$ sudo swapon /path/to/swapfile
```

These commands will generate a 1 GB swap file (1024 x 1 M = 1 GB) at `/path/to/swapfile`, which you should change to the location of your swapfile. To make the swap file automatically enabled on boot, add a new line to your `/etc/fstab` file:

```
/path/to/swapfile none swap defaults 0 0
```

You will see the additional memory (as swap) in the output of the `free` command:

```
$ free
total used free shared buffers cached
Mem: 448688 436960 11728 0 6776 395392
```