My Quest to Build the Ultimate Music Player

Andrew Kelley

HACKERMONTHLY Issue 51 August 2014

Curator
Lim Cheng Soon

Contributors
Andrew Kelley
Yu Jiang Tham
Chris Loukas
Hadi Hariri
Robert Muth
Brian Green

Proofreaders
Emily Griffin
Sigmarie Soto

Ebook Conversion
Ashish Kumar Jha

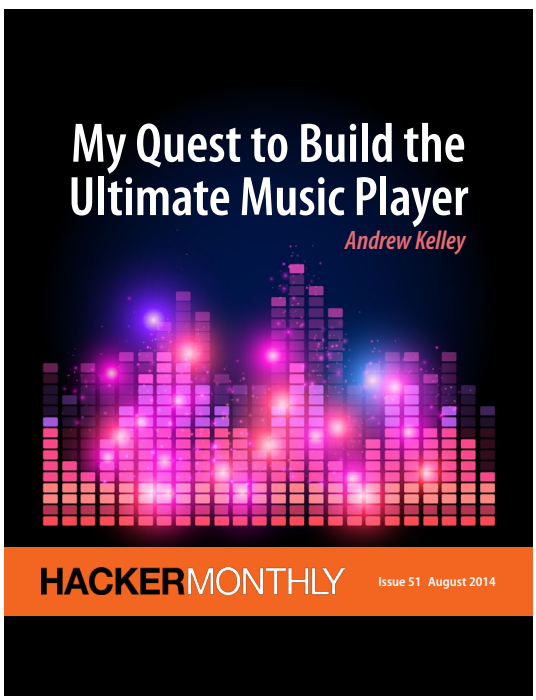
Printer
MagCloud

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be "anything that gratifies one's intellectual curiosity." Every month, we select from the top voted articles on Hacker News and print them in magazine format.
For more, visit *hackermonthly.com*

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

Published by
Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.



Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-51*

My Quest to Build the Ultimate Music Player

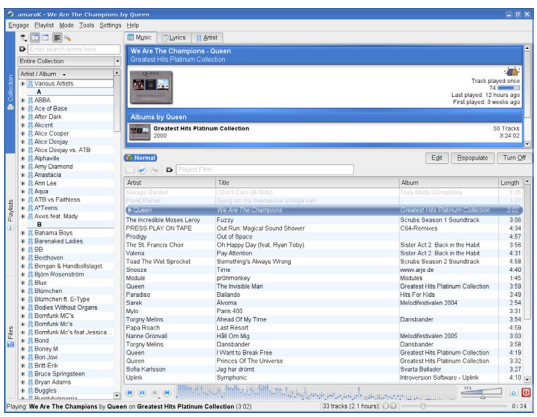
By ANDREW KELLEY

Over the past few years, I have been slowly but surely building my own music player. It's been a wild ride. The codebase has radically changed several times, but it is always converging on a better music listening experience.

In this article my goal is to take you along for the ride.

I <3 Amarok 1.4

Back in 2009, my music player of choice was Amarok 1.4. This was by far the best music player I had ever used on Windows, Mac, or Linux, especially when combined with the wonderful ReplayGain plugin. Here's a screenshot:



One way you can tell how much people loved this music player is by looking at the comments on the release blog articles for Amarok 2.0, which rebuilt the player from scratch and took it in a completely different direction. Arguably, they should have picked a different project name and logo. Look at some of these comments, how angry and vitriolic they are:

- 2.0 [hn.my/amarok20]
- 2.1 [hn.my/amarok21]
- 2.2 [hn.my/amarok22]

Even now, 4 years later, the project is at version 2.8 and the release name is titled "Return To The Origin":

> *Amarok 1.8 is titled "Return To The Origin" as we are bringing back the polish that many users loved from the original 1.x series!*

That should give you an idea of how much respect Amarok 1.4 commanded.

Even so, it was not perfect. Notably, the ReplayGain plugin I mentioned above had several shortcomings. Before I get into that, however, let me take a detour and explain what ReplayGain, or more generally, loudness compensation, is and some of its implications.

A Short Explanation of Loudness Compensation

Have you ever seen this 2-minute video explaining the Loudness War? [hn.my/loudnesswar]

The video demonstrates a trend in digital audio mastering where songs are highly compressed to sound louder, and how this can compromise the integrity of the music.

While thinking about building a music player, we're not going to make moral judgments about whether or not compression is ruining music for everybody. If users want to listen to highly compressed music, that's a valid use case. So we have to consider a music library which contains both compressed songs and dynamic songs.

Here is a song called The Happiest Days of Our Lives by Pink Floyd, mastered in 1979:

Here is a song called Saying Sorry by Hawthorne Heights, mastered in 2006:

It is immediately obvious that the second one is much louder than the other. So what happens when they are played one after the other in a music player?

When the quieter song comes on first, the user reaches for the volume knob to turn it up so they can hear. Oops. When the next song begins, a surge of adrenaline shoots through the user's body as they scramble to turn the volume down. This goes beyond poor usability; this problem can cause hearing loss.

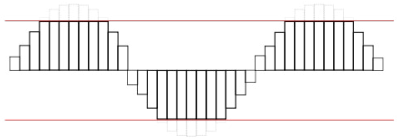
The solution is to analyze each song before playing it to figure out how "loud" it sounds to humans. Then the music player adjusts the playback volume of each track to compensate for the perceived loudness. This way, the user does not have to adjust the volume for each track that comes on.

The idea is simple enough, but it poses a few subtle challenges.

For one, the loudness of an individual track might be different than the loudness of the album as a whole. A complete loudness compensation solution has to take this into account, both during scanning and playback.

An even trickier problem is avoiding **clipping**. Music is composed of samples which have a fixed range. For example in floating point format, samples can be between 0.0 and 1.0. Even quiet songs usually have some samples which peak at 1.0, for example on the drums. But we need to turn the volume up on these quiet songs to make them sound as loud as the highly compressed ones.

If we naïvely increased the volume on such a song, we would end up with something like this:



The grey bars above the red lines represent clipping. This causes distortion and generally sounds awful.

The solution is not to increase the volume of the quiet song, but to decrease the volume of the loud song. In order to do this, we introduce an amount called pre-gain. All songs are turned down by this amount, which gives us the headroom we need to turn the quieter ones back up.

It's not a perfect solution though.

The lower the pre-gain, the more the music player will sound quieter than other applications on the computer. The higher the pre-gain, the more likely that there is not enough headroom to increase the volume of a quiet song enough.

In 2010, the European Broadcasting Union introduced a new standard called R128. This standard outlines a strategy for analyzing media and determining how loud it is. There is a motion to make ReplayGain 2.0 use this standard.

Shortcomings of Amarok 1.4

As much as I loved Amarok 1.4, it did not even attempt to address these loudness issues. There is no built-in loudness compensation.

The ReplayGain plugin I mentioned earlier was great, but it was limited in usefulness:

- It had to scan every time the playlist updated; it didn't cache the data.
- Each format that you wanted to scan had a different command-line utility which had to be installed. This means that the set of songs that Amarok 1.4 could play was completely different than the set of songs that it could scan.
- It applied the volume changes on a gradient instead of instantly, and timing was not precise. This means that it might erroneously turn up the loudness far too high in the transition time to the next track. This behavior was distracting and sometimes ear-piercingly painful.
- You had to manually decide between track and album mode. This is a pointless chore that the music player should do automatically. Here's a simple algorithm:
  - If the previous item in the playlist is the previous item from the same album, or the next item in the playlist is the next item from the same album, use the album ReplayGain information.
  - Otherwise, use the track ReplayGain information.

Aside from the loudness compensation, I had a couple other nits to pick:

- Dynamic Mode was a useful feature that could continually play random songs from the library. But the random selection was too random; it would often queue the same song within a short period of time.
- If the duration tag was incorrect in a song, or if in was a variable rate MP3, the song would seemingly end when the song had not yet gotten to the end. Or in other words, the reported duration was incorrect and seeking would be broken.

I've spent some time criticizing, now let me be more constructive and actually specify some features that I think music players should have.

My Laundry List of Music Player Features

Loudness Compensation using the same scanner as decoder

This is absolutely crucial. If you want to solve the loudness compensation problem, the set of songs which you can decode and play back must be the same set of songs which you can scan for loudness. I should never have to manually adjust the volume because a different song or album came on.

Ideally, loudness scanning should occur lazily when items are added to the play queue and then the generated values should be saved so that the loudness scanning would not have to be repeated.

Do not trust duration tags

A music player already must scan songs to determine loudness compensation values. At the same time, it should determine the true duration of the file and use that information instead of a tag which could be wrong.

If my friends come over, they can control the music playback

Friends should be able to upload and download music, as well as queue, skip, pause, and play.

Ability to listen to my music library even when I'm not home

I should be able to run the music player on my home computer and listen to a real-time stream from work, for example.

Gapless Playback

Many albums are created in order to be a listening experience that transcends tracks. When listening to an album, songs should play seamlessly and without volume changes at the seams. This means that loudness scanning must automatically take into account albums.

Robust codec support

You know how when you need to play some obscure video format, you can always rely on VLC to play it? That must be true for the ultimate music player as well. A music player must be able to play music. If you don't have a wide range of codecs supported, you don't have a music player.

Keyboard Shortcuts for Everything

I should be allowed to never touch the mouse when operating the music player.

Clean up my messy files

One thing that Amarok 1.4 got right is library organization. It offered a powerful way to specify the canonical location for a music file, and then it had an option to organize properly tagged music files into the correct file location.

I don't remember the exact format, but you could specify a format something like this:

```
%artist/%album/%track %title%extension
```

Filter Search

There should be a text box where I can type search terms and instantly see the search results live. And it should ignore diacritics. For example, I could type "jonsi ik" and match the song Boy Lilikoi by Jónsi.

Playlist Mode that Automatically Queues Songs

Some names for this feature are:

- Dynamic Mode
- Party Mode
- DJ Mode

The idea is that it automatically queues songs — kind of like a real-time shuffle — so that you don't have to manually decide what to listen to.

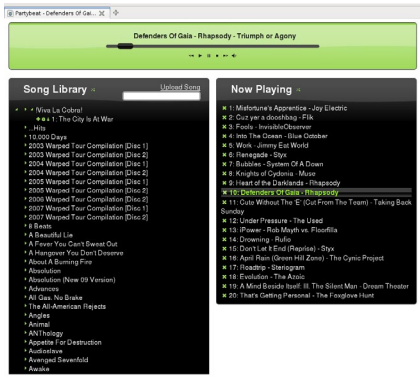
One common flaw found in many players is using a truly random algorithm. With true randomness, it will frequently occur that a song which has recently been randomly chosen will be randomly chosen again.

A more sophisticated algorithm weights songs by how long it has been since they have been queued. So any song would be possible to be queued, but songs that have not been queued recently are much more likely to be queued. Queue date is chosen rather than play date because if a song is queued and the user skips the song, this should still count in the weight against it being chosen again.

"PartyBeat"

It would be a long time before my wish list of features would become a reality. Meanwhile, back in college my buddy made a fun little project [hn.my/partybeat] which served as a music player that multiple people could control at the same time with a web interface. He installed it in my and my roommate's apartment, and the three of us used it in our apartment as a shared jukebox; anarchy deciding what we would listen to while we worked on our respective jobs, homework, or projects. We dubbed it "PartyBeat."

Here's a screenshot:



This project used Django and xmms2 and had a bug-ridden, barren, and clunky web-based user interface. It was so lacking compared to my usual Amarok 1.4 experience, yet somehow I could not give up the "shared jukebox" aspect. It was simply too fun to listen to music together, regardless of the interface.

So finally I decided to build the ultimate music player. It would still have a web-based interface, but it would behave like a native application — both in feel and in responsiveness. It should be nice enough that even when you want to listen alone it would still be your go-to player of choice.

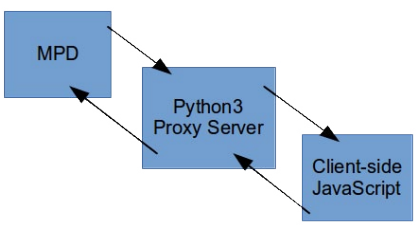
Fumbling Around with Technology

In early 2011 I started investigating what technology to use to build this thing. I knew that I wanted a backend which could decode many audio formats, do gapless playback, and provide some kind of interface for a web server to control it.

I tinkered a bit with Qt and the Phonon framework, but I didn't get as far as having a web interface controlling it.

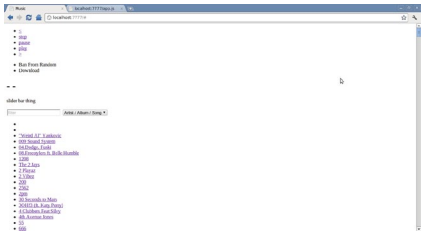
Eventually I stumbled upon Music Player Daemon. At the time this seemed like a perfect fit, especially since the XMMS2 wiki admitted that if they had known that MPD existed when they started the project, they would probably have just used it. MPD is a service — it has a config file which tells it, among other things, the location of your music library, and then it runs in the background, listening on a port (typically 6600), where you can issue commands via the protocol telling it to pause, play, skip, queue, unqueue, and all that jazz.

The first iteration of "PartyBeat2" was a small Python 3 server which was merely a proxy between the client-side JavaScript code and MPD, as well as a file server to serve the client-side HTML and JavaScript.

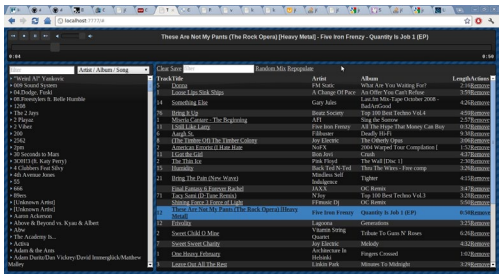


At this point I had a basic proof-of-concept. However, progress slowed for a few months as I embarked on a 12-day hiking trip followed immediately by the first day of work at Amazon, my first out-of-college job.

After a short hiatus, I revisited the project. This was right when socket.io was getting a lot of hype, and it seemed like the perfect fit for my design. Also I had just given CoffeeScript a real chance after snubbing it initially. So I ported over the proxy/file server to Node.js and got a prototype working:



A week of iterating later, I had the basics of a user interface, and a name:



I named it Groove Basin, after the Sonic the Hedgehog 3 Azure Lake Remix by Rayza. As homage to the original project, I picked a JQuery UI theme for the UI, except this time I chose Dot Luv.

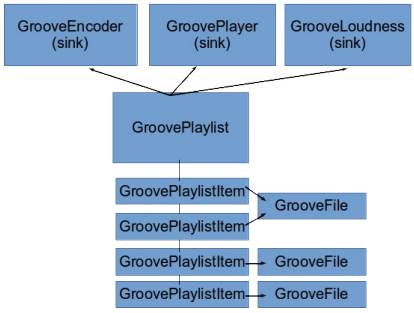
Building a Music Player Backend

So, how hard could it be to build my own music player backend? Seems like it would be a matter of solving these things:

- Use a robust library for audio decoding. How about the same one that VLC uses?
- Support adding and removing entries on a playlist for gapless playback.
- Support pause, play, and seek.
- Per-playlist-item gain adjustment so that perfect loudness compensation can be implemented.
- Support loudness scanning to make it easy to implement, for example, ReplayGain.
- Support playback to a sound device chosen at runtime.
- Support transcoding audio into another format so a player can implement, for example, HTTP streaming.
- Give raw access to decoded audio buffers just in case a player wants to do something other than one of the built-in things.
- Try to get other projects to use it to benefit from code reuse.
  - Make the API generic enough to support other music players and other use cases.
  - Get it packaged into Debian and Ubuntu.
  - Make a blog post about it to increase awareness.

After reading up a little bit on the insane open-source soap-opera that was the forking of libav from ffmpeg (here are two sides to the story: libav side, ffmpeg side), I went with libav simply because it is what is in the Debian and Ubuntu package managers, and one of my goals is to get this music player backend into their package managers.

Several iterations later, I now have libgroove, a C library with what I think is a pretty solid API. How it works:



The API user creates a GroovePlaylist which spawns its own thread and is responsible for decoding audio. The user adds and removes items at will from this playlist. They can also call pause, play, and seek on the playlist. As the playlist decodes audio, where does the decoded audio go? This is where those sinks come in.

A **sink** is a metaphor of a real-life sink that you would find in a bathroom or kitchen. Sinks can fill up with water, and unless the water is drained the sink will continue to fill until it overflows. Likewise, in audio processing, a sink is an object which collects audio buffers in a queue.

In libgroove, decoded audio is stored in reference-counted buffer objects and passed to each connected sink. Each sink does whatever processing it needs to do and then calls "unref" on the buffer. Typically each sink will have its own thread which hungrily waits for buffers and devours them as fast as possible. However the playlist is also decoding audio as fast as possible and pushing it onto each sink's queue. It is quite possible, that a sink's queue fills up faster than it can process the buffers. When the playlist discovers that all its sinks are full, it puts its thread to sleep, waiting to be woken up by a sink which has drained enough.

libgroove provides some higher-level sink types in addition to the basic sink. Each higher level sink runs in its own thread and is built using the basic sink. These include:

- **playback sink**: opens a sound device and sends the decoded audio to it. This sink fills up with events that signal when the sink has started playing the next track, or when a buffer underflow occurs.
- **encoder sink**: encodes the audio buffers it receives and fills up with encoded audio buffers. These encoded buffers can then be written to a file or streamed over the network, for example.
- **loudness scanner sink**: uses the EBU R 128 standard to detect loudness. This sink fills up with information about each track, including loudness, peak, and duration.

The API is designed carefully such that even though the primary use case is for a music player backend, libgroove can be used for other use cases, such as transcoding audio, editing tags, or ReplayGain scanning. Here is an example of using libgroove to for a simple transcode command line application:

```
/* transcode one or more files into one output file */

#include <groove/groove.h>
#include <groove/encoder.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

static int usage(char *arg0) {
    fprintf(stderr, "Usage: %s file1 [file2 ...] --output outputfile [--bitrate 320] [--format name]
[--codec name] [--mime mimetype]\n", arg0);
    return 1;
}
```

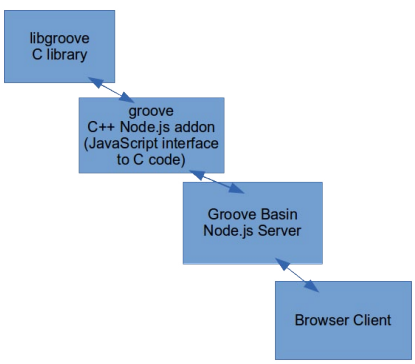
```
int main(int argc, char * argv[]) {
    // arg parsing
    int bit_rate_k = 320;
    char *format = NULL;
    char *codec = NULL;
    char *mime = NULL;

    char *output_file_name = NULL;

    groove_init();
    atexit(groove_finish);
    groove_set_logging(GROOVE_LOG_INFO);
    struct GroovePlaylist *playlist = groove_playlist_create();

    for (int i = 1; i < argc; i += 1) {
        char *arg = argv[i];
        if (arg[0] == '-' && arg[1] == '-') {
            arg += 2;
            if (i + 1 >= argc) {
                return usage(argv[0]);
            } else if (strcmp(arg, "bitrate") == 0) {
                bit_rate_k = atoi(argv[++i]);
            } else if (strcmp(arg, "format") == 0) {
                format = argv[++i];
            } else if (strcmp(arg, "codec") == 0) {
                codec = argv[++i];
            } else if (strcmp(arg, "mime") == 0) {
                mime = argv[++i];
            } else if (strcmp(arg, "output") == 0) {
                output_file_name = argv[++i];
            } else {
                return usage(argv[0]);
            }
        } else {
            struct GrooveFile * file = groove_file_open(arg);
            if (!file) {
                fprintf(stderr, "Error opening input file %s\n", arg);
                return 1;
            }
            groove_playlist_insert(playlist, file, 1.0, NULL);
        }
    }
    if (!output_file_name)
        return usage(argv[0]);

    struct GrooveEncoder *encoder = groove_encoder_create();
    encoder->bit_rate = bit_rate_k * 1000;
    encoder->format_short_name = format;
```

```
    encoder->codec_short_name = codec;
    encoder->filename = output_file_name;
    encoder->mime_type = mime;
    if (groove_playlist_count(playlist) == 1) {
        groove_file_audio_format(playlist->head->file, &encoder->target_audio_format);

        // copy metadata
        struct GrooveTag *tag = NULL;
        while((tag = groove_file_metadata_get(playlist->head->file, "", tag, 0))) {
            groove_encoder_metadata_set(encoder, groove_tag_key(tag), groove_tag_value(tag), 0);
        }
    }

    if (groove_encoder_attach(encoder, playlist) < 0) {
        fprintf(stderr, "error attaching encoder\n");
        return 1;
    }

    FILE *f = fopen(output_file_name, "wb");
    if (!f) {
        fprintf(stderr, "Error opening output file %s\n", output_file_name);
        return 1;
    }

    struct GrooveBuffer *buffer;

    while (groove_encoder_buffer_get(encoder, &buffer, 1) == GROOVE_BUFFER_YES) {
        fwrite(buffer->data[0], 1, buffer->size, f);
        groove_buffer_unref(buffer);
    }

    fclose(f);

    groove_encoder_detach(encoder);
    groove_encoder_destroy(encoder);

    struct GroovePlaylistItem *item = playlist->head;
    while (item) {
        struct GrooveFile *file = item->file;
        struct GroovePlaylistItem *next = item->next;
        groove_playlist_remove(playlist, item);
        groove_file_close(file);
        item = next;
    }
    groove_playlist_destroy(playlist);

    return 0;
}
```

Note that this code contains no threading. Even so, because of the way libgroove is designed, when this app runs, one thread will work on decoding the audio while the main thread seen in this code will work on writing the encoded buffers to disk.

Once I had this backend built, I needed to use it in Groove Basin, which you may recall is a Node.js app. To do this I built a native add-on node module called groove. It uses libuv and v8 to interface between C and Node.js. I wrote the majority of this code at Hacker School, an experience which I highly recommend.

With the groove node module complete, the new architecture looked like this:



No longer did Groove Basin need to run a third party server to make everything work — just a single Node.js application with the correct libraries installed. This put me in control of the audio backend code which meant that I had the power to make everything work exactly like I wanted it to.

Packaging

Nothing turns away potential users faster than a cumbersome install process. I knew that I had to make Groove Basin easy to install, so I took several steps to make it so.

One thing I did was bundle some of the harder-to-find dependencies along with it. Specifically, libav10, libebur128, and SDL2. This way if the user is on a computer that does not have those packages readily available, they may still install libgroove.

This convenience is less desirable than relying on existing system dependencies, however, so if the configure script detects system libraries, it happily prefers them.

Next, I made a libgroove PPA for Ubuntu users. This makes installing libgroove as easy as:

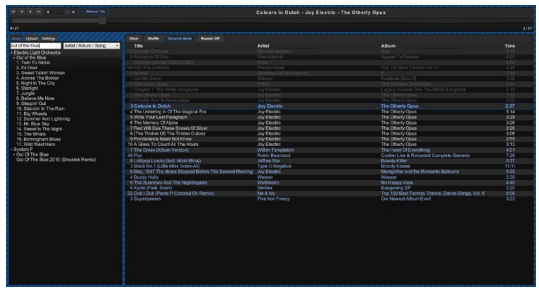
```
sudo apt-add-repository
ppa:andrewrk/libgroove
sudo apt-get update
sudo apt-get install lib-
groove-dev libgrooveplayer-dev
libgrooveloudness-dev
```

Then I joined the Debian multimedia packaging team. This team is dedicated to making Debian a good platform for audio and multimedia work. They kindly accepted me and coached me while I worked on packaging up libebur128 and libgroove for Debian. After a few back and forths, a libebur128 Debian package is ready to be installed from testing, and a libgroove Debian package can be installed from experimental. Once the libav10 transition is complete, libgroove can be submitted to unstable, where it will move into testing, and then finally be released to all of Debian!

After a few more months of progress, I'd like to package up Groove Basin itself. This way, the entire installation process could be just an `apt-get install` away.

Conclusion

3 years, 6 months from git init and Groove Basin is still under active development. Here's what the UI looks like today:



Some of the features that it provides are:

- Fast, responsive UI. It feels like a desktop app, not a web app.
- Dynamic playlist mode which automatically queues random songs, favoring songs that have not been queued recently.
- Drag and drop upload. Drag and drop playlist editing. Rich keyboard shortcuts.
- Lazy multi-core EBU R128 loudness scanning (tags compatible with ReplayGain) and automatic switching between track and album mode. "Loudness Zen"
- Streaming support. You can listen to your music library — or share it with your friends — even when you are not physically near your home speakers.
- MPD protocol support. This means you already have a selection of clients which integrate with Groove Basin. For example MPDroid.

- Last.fm scrobbling.
- File system monitoring. Add songs anywhere inside your music directory and they instantly appear in your library in real time.
- Supports GrooveBasin Protocol on the same port as MPD Protocol — use the "protocol upgrade" command to upgrade.

If you like you can try out the web interface client of Groove Basin on the live demo site. It will probably be chaotic and unresponsive if there is a fair amount of traffic to this blog post, as it's not designed for a large number of anonymous people to use it together; it's more for groups of 10 or less people who actually know each other in person.

The roadmap moving forward looks like this:

1. Tag Editing
2. Music library organization
3. Accoustid Integration
4. Playlists
5. User accounts / permissions rehaul
6. Event history / chat
7. Finalize GrooveBasin protocol spec

Groove Basin still has lots of issues but it's already a solid music player and it's only improving over time.

Feel free to star or watch the Groove Basin GitHub repository if you want to keep track of progress. [github.com/andrewrk/groovebasin] ■

Andrew Kelley is born and raised in Phoenix, Arizona. He learned programming from wanting to make video games. Andrew is in love with free and open source software.

Reprinted with permission of the original author.
First appeared in *hn.my/music* (andrewkelley.me)

Build Your Own Drink Mixing Robot

By YU JIANG THAM

I built a robot that mixes drinks named Bar Mixvah. It utilizes an Arduino microcontroller switching a series of pumps via transistors on the physical layer, and the MEAN stack (MongoDB, Express.js, Angular.js, Node.js) and jQuery for the frontend and backend. In this article, I'll teach you how I made it. You can follow along and build one just like it! I've also put the 3d model (blender), stl files, and the code up on GitHub for you guys to download. See the link at the bottom of the article. Here's the video of the robot: [hn.my/drinkbotdemo]

First, a little bit more about the robot. The entire thing costs approximately $180 to make. All of the parts are 3d printed, so you'll need a 3d printer to build this. I used the MakerBot Replicator 2X, but any 3d printer should do. Total time to print the pieces is about 18 hours, depending on your settings, and assembly wiring. Here's a list of parts that need to be purchased:

- 5x 12V DC peristaltic pumps
- 11x 5/16" steel square 12" rods
- Clear tubing
- Arduino Nano
- 5x TIP120 w/ Diodes
- 400-point breadboard and jumper wire
- 5x 2.2kOhm resistor
- 4x #6-32 2" machine screws
- 10x #4-40 3/4" machine screws
- 12V power supply rated at (or greater than) 1.5A - or you can use an old laptop power supply (as long as it's 12V DC).
- 5.5mm x 2.1mm coaxial power connector (female) - or if you're using a laptop power supply, 5.5mm x 2.5mm
- Male pin connectors
- Female housing for the male pin connectors

Other tools required for the job are: a hacksaw to cut two of the 12" rods in half, a wire stripper, soldering iron, and solder to connect the wire to the pin connectors and coaxial power connector, and a multimeter to check your work.

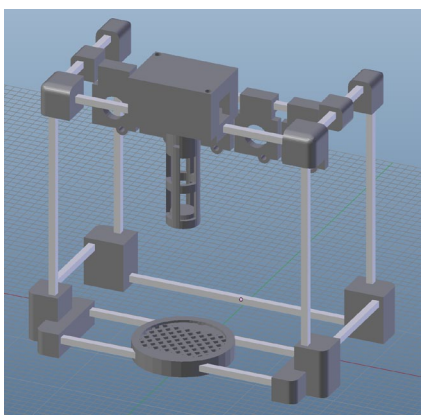
Design

Bar Mixvah is designed to use a system of 5 peristaltic pumps that are switched by 5 bipolar junction transistors (TIP120), all controlled by an Arduino, which itself is controlled by the Johnny-Five package on the node.js/express web server that is running on your laptop/Windows tablet (or maybe Raspberry Pi? I haven't tried). Having it on a web server allows users to order from any device, be it a phone, tablet, or other laptop that can connect to your WiFi access point's internal network. Practicality-wise, maybe it's not necessary. However, in my experience, people seem to enjoy ordering from a tablet that they're holding in their hands more

than a stationary screen attached to the robot.

The physical design of Bar Mixvah is around the usage of 5/16“x12” steel rods. I chose this length and size because they’re sturdy, readily available at your local hardware store, and not too big or small. They’re also relatively cheap at ~$2-3 per piece, depending on where you buy from. The problem with 3d printing is that it’s goddamn slow. If you want to build a medium sized robot like this one, it would take days to print all of the necessary parts. In fact, you don’t even need to print these parts; you could fasten them together using plenty of other methods. However, I don’t have access to a metal shop, am a terrible welder, and wanted a friendly looking robot, so I chose this combination of 3d printing the joints and connecting them via metal shafts.

Here’s a screenshot of the 3d model, which, fortunately, looks exactly like the real thing after I finished building it. Ah, the miracles of science!



Printing the Parts

The stl files can mostly be printed in the orientation that they are in, however two files should be rotated 180 degrees on the x-axis so that they can be printed without major supports. These two pieces are `Center - Board Cover - Top.stl` and `Center - Common Drink Channel.stl`. Additionally, `Center - Pump Holder.stl` should be printed flat by rotating it 90 degrees so that no support pieces are needed. You will need to turn on printing with supports to ensure that the holes where we will be inserting the 5/16" steel rods are printed to the right size.

One more thing that has been brought to my attention: the .stl files may be 10x smaller than they should be in your 3d printing software. If that’s the case, you will need to scale up the objects 10x in all dimensions. It seems to happen regardless of the 3d software that I use when I convert to .stl. No idea why.

The Peristaltic Pumps

What is a peristaltic pump? If the word sounds familiar to you, it’s because you most likely heard of peristalsis in one of your biology classes at some point in time. Peristalsis is the system that your body uses to swallow food. Your throat muscles contract and relax in a way to create a waveform that pushes food in your throat down into your stomach. Peristaltic pumps work on the same principle, albeit with a slightly different execution. The clear plastic tube extends through the pump, and rollers propelled by a DC motor create a waveform that pushes liquid through the tube. Peristaltic pumps are safe and hygienic because liquid never actually contacts any part of the pump; only the plastic tubing is ever in contact with the liquid.

The peristaltic pumps come with some very short plastic tubing. This is obviously inadequate for our current application, so we’ll have to replace the plastic tubing. This requires us to take apart the pump. Fortunately, this is not a hard thing. Instead of trying to explain it, you can view the following video to figure out how it is done. [hn.my/peripump]

Soldering

Before connecting any wires, you’ll want to do all of the soldering. You’ll have to solder the 5.5mm x 2.1mm coaxial power connector to two jumper wires, one for the positive lead and one for the negative lead. Plug in your 12V DC power supply to the wall, then plug the coaxial power connector into the DC power supply. Use your multimeter to find out which lead is positive and negative by placing the probes on two of the leads until you find that the multimeter says 12V; those are your positive and negative leads (if it says -12V, then you’ve got the positive and negative leads switched). Unplug the coaxial power connector. Strip two wires and solder one to each of the coaxial power connector’s leads. After you’re done soldering, wrap any exposed metal around the leads in electrical tape.

Next, you’ll want to solder wires to the leads of the peristaltic pumps. The positive lead of the pump should be labeled, so you should not need to guess. If it is not labeled, you will just need to make note of which way the pump is turning and make sure that all of them are turning in the same

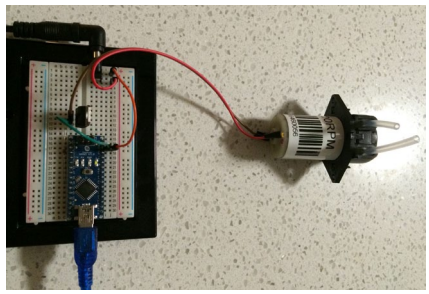
direction. Don't worry about the polarity of the leads on the pump breaking anything, since connecting them backwards will just make the pump go in the opposite direction. However, I want to emphasize that you'll probably want to ensure all of the pumps are pumping clockwise (if they are facing you; you can see through the tiny circle in the middle which direction they are pumping when turned on). After this is done, once again wrap the leads in electrical tape.

Wiring

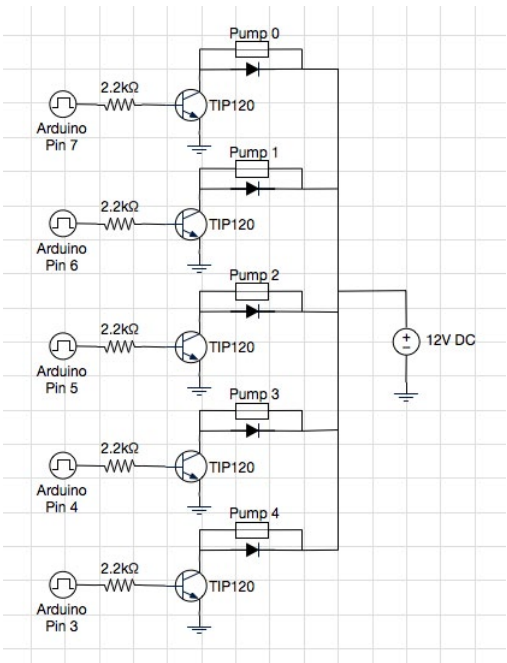
Here's where it gets a little bit tricky. The actual wiring is not too complicated, but it requires a little bit of finesse due to the confines of space that we are working with. Since we are fitting everything on a single breadboard, we need to ensure everything is placed in the right spot.

In case you haven't used a breadboard in a while, each of the numbers running down the breadboard indicate an individual node. The center divides the two sides, so they are separate nodes. The (+) rail running up the left and right side of the breadboard is one node per side, and it is the same with the (-) rail.

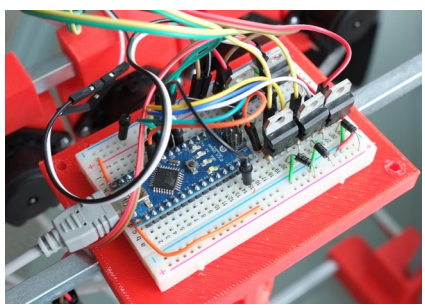
The first thing that you should do before getting any wiring done is to hook up your pumps individually and ensure they are all working. The photo below shows a little bit more complex of a circuit. To check if it's working, you can just connect the coaxial power connector and pump on a breadboard and plug in the power and ensure that the pump works.



Here's the wiring diagram for the robot. As you can see, it's relatively simple:



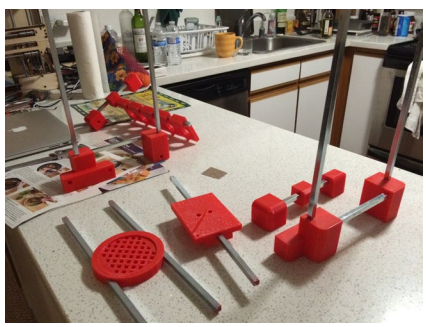
The tough part is that there is not much space, so you may need to have a set of needle-nosed pliers ready to put some of the things in. I recommend adding everything except transistors in first. Here's how my breadboard looked like after I finished wiring it up:



Obviously, yours may look slightly different. However, I recommend placing the Arduino's nano so that its USB port is on the edge of either side of the breadboard.

Assembly

After printing all of the pieces, remember which piece is which based on the 3d model. Remove all of the support pieces from the 3d printed items with pliers and/or a flathead screwdriver. It is likely that it will be a very tight fit for the steel rods, so you'll need to push the rods in with a lot of force. I recommend using gloves.



Also, the order in which you assemble the robot IS important. Here's the order that I used to assemble. Basically, you need to remember to assemble the middle parts first before connecting the left and right sides to them.

1. Insert all 5 of the pump holders onto one of the steel rods
2. Insert the drink tray into the center of two of the steel rods
3. Insert the breadboard holder (bottom piece) into the center of one steel rod
4. Assemble the left side, then assemble the right side
5. Insert the steel rods for the drink tray, pump holders, and

breadboard holder into the left side

6. Connect all of the parts of the right side
7. Insert each of the pumps into a pump holder and screw them in with two #4 screws each
8. Attach the center channel to the top center section using two #6 screws
9. Tape the breadboard to the center of the top-middle section, ensuring that the top piece of this section will fit over it
10. Wire everything up based on the circuit schematic
11. Place the top piece (labeled #BarMixvah in the photo below) over the breadboard, moving wires around until everything fits snugly inside
12. Insert two #6 screws through the screw hole and tighten the nuts at the bottom to secure it in place

Software Design

In this section, I will be discussing the software design of the robot, as well as the considerations that I took in designing it. I'm assuming that you have some working knowledge of JavaScript and MongoDB. I realize that the Angular.js code that is written here may not be the most efficient, since the whole point of me building the robot was to learn Angular (and also make something awesome). Therefore, if you spot any changes that need to be made to the code to increase efficiency or whatnot, please let me know in the comments or send a pull request to the Bar Mixvah GitHub.

Why the MEAN Stack

So, before I proceed, a few people have asked why we need so much technology to do something so simple as mixing drinks. What is the MEAN stack? It consists of MongoDB, Express.js, Angular.js, and Node.js. Everything works beautifully together because it's all JavaScript (and MongoDB, which uses BSON, similar to JavaScript's JSON). Why do we need the MEAN stack? Well, we don't need it. But it makes things easier, so that's why I'm using it.

- MongoDB is used to store all of the drink and pump information
- Express.js is the web server so that your WiFi-connected devices can access the interface
- Angular.js is used on the front end to filter drinks based on what ingredients have been selected
- Node.js is used on the backend to communicate with the Arduino

Preparing Your Arduino

The Arduino Nano talks to the Johnny-Five node.js package via the Standard Firmata protocol. You will need to flash Standard Firmata onto your Arduino:

1. Download the Arduino software
2. Plug in the Arduino to your USB port on your computer
3. Open the Arduino software
4. Go to File > Examples > Firmata > StandardFirmata
5. Click the Upload button in the top-left

Starting the App

Bar Mixvah requires node.js, MongoDB, and git. Follow the respective links to download and install them if you haven't already. The commands here are run in the Mac OS or Linux shell; I don't have much experience with the Windows command line, but I believe most of the stuff is the same (except you should remove sudo if you see it, and run your command prompt as

an administrator). To start off, clone the Bar Mixvah GitHub repository onto your local hard drive:

```
git clone https://github.com/ytham/barmixvah.git
```

cd into the directory and install all node.js packages:

```
cd barmixvah
npm install
```

This will install all of the packages required for the node.js app to run. Before you can run it, you must start mongodb:

```
sudo mongod
```

In a new terminal window, cd to the barmixvah directory again. Plug in the Arduino to your computer via USB. You can now start the robot app:

```
node app.js
```

Congrats! It should now be up and running. You can now point your web browser to http://localhost:3000 to check it out. The rest of this post will go towards explaining the code for the robot. If you haven't yet set up the Arduino, you can read the next section to set up the UI without having it connected to an Arduino for testing purposes.

Debugging the App Without the Arduino

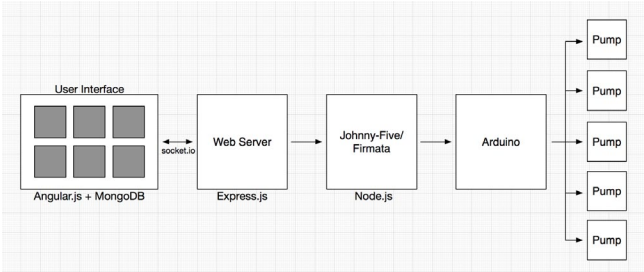
The app can be started without an Arduino by commenting out the following section:

```
> public/javascripts/robot/backend.js
var board, pump0, pump1, pump2, pump3, pump4;
/*
var five = require('johnny-five');

board = new five.Board();
board.on('ready', function () {
 // Counting down pins because that's the orientation
 // that my Arduino happens to be in
 pump0 = new five.Led(7);
 pump1 = new five.Led(6);
 pump2 = new five.Led(5);
 pump3 = new five.Led(4);
 pump4 = new five.Led(3);

 board.repl.inject({
  p0: pump0,
  p1: pump1,
  p2: pump2,
  p3: pump3,
  p4: pump4
 });

 console.log("\033[31m[MSG] Bar Mixvah Ready\033[91m");
});
*/
```

This will prevent Johnny-Five from initializing the Arduino. This is useful for testing UI or backend code on the go if you're not tethered to an Arduino.

General Flow

Here's a pretty simplified design that gives you a general picture of how the code works with the physical elements:



When a user interacts with the UI, Angular.js (or some custom JavaScript/jQuery) will change what is displayed on the screen. If the user decides to make a drink with the Make button, the selected drink JS object and drink size are passed to the backend via socket.io. From there, the app passes the data to the Johnny-Five package and communicates when to start/stop the array of pumps.

Drink and Pump Schemas

The drink schema that will be saved into MongoDB (via the Mongoose node.js package) is specified below:

```
> models/Drink.js
exports.DrinkSchema = new Mongoose.Schema({
 name: { type: String, required: true },
 image: { type: String, required: false },
 ingredients: [{
  name: String,
  amount: Number
 }]
});
```

The drink objects have a name, image, and an array of ingredient objects that contain their name and relative amount. Ingredient amounts are relative and unitless because then the user can specify the drink size and get the actual amount of each ingredient to pump based on the specified drink size.

Here is the pump schema:

```
> models/Pump.js
exports.PumpSchema = new Mongoose.Schema({
 label: { type: String, unique: true, sparse:
true, required: true },
 ingredients: [{
  label: String,
  ingredient: String
 }]
});
```

We have one pump object that contains an array of pumps that have a label (such as "pump0", "pump1", etc…) and an ingredient associated with that pump. In this case, we use the label because order is important and we want to ensure that the pumps are in the correct order regardless of how they are modified. When any pump is updated via the UI, the entire pump object that contains all of the (up to five) pumps is updated in MongoDB. This keeps things consistent and ensures that pumps are updated correctly.

```
> routes/index.js
exports.updatePump = function (Pump) {
 return function (req, res) {
  Pump.findOneAndUpdate({ _id: req.body._id },
   {
    ingredients: req.body.ingredients
   },
  function (err, pump) {
   if (pump == null) {
    Pump.create(req.body);
    pump = req.body;
   }
   res.send(pump);
  });
 }
}
```

Choosing Pumps

The pump system is set up so that as soon as the any of the pump ingredients are changed, the entire pumps object (containing all of the individual pump ingredients) is changed. The ng-click directive in this case calls two functions. One function saves the pumps object by overriding the previous pumps object, the other figures out the number of duplicate ingredients and writes the number of duplicates that are checked at other times (such as when the Make button is pressed). The reason why we don't just check pumps for duplicates immediately is if, say you are a user and you want to move "Orange Juice" from pump0 to pump2. You might change pump2 to "Orange Juice" first, but if that throws an error since "Orange Juice" is also currently on pump0, that is not a very good user experience.

```
> views/index.jade
div.pumpContainer(ng-repeat="pump in pumps.
ingredients")
 select.mixers(ng-change="savePumpValue($index);
writeNumDuplicates()", ng-model="pump.ingredi-
ent", ng-options="i for i in ingredientsList")
```

The savePumpValue function sends a post request with the $scope.pumps object. The pumps object is data bound to the view via Angular, so the changes in the dropdown are automatically modified on the front end, but we just need to save it into the database so that when the user refreshes the page, they get the same pumps instead of having to start over.

```
> public/javascripts/controller/DrinkController.js
$scope.savePumpValue = function (pumpNumber) {
 $http.post('/updatepump.json', $scope.pumps).
success(function (data) {
  if (data) {
   console.log(data);
  }
 });
};
```

From here, the web server receives the HTTP POST request via the updatepump.json endpoint.

```
> app.js
var routes = require('./routes/index');
...
app.post('/updatepump.json', routes.
updatePump(Pump));
```

And then the updatePump function is run, creating a pump if there is none, and updating the current pumps object if it already exists.

```
> routes/index.js
exports.updatePump = function (Pump) {
 return function (req, res) {
  Pump.findOneAndUpdate({ _id: req.body._id },
  {
   ingredients: req.body.ingredients
  },
  function (err, pump) {
   if (pump == null) {
    Pump.create(req.body);
    pump = req.body;
   }
  res.send(pump);
  });
 }
}
```

Pump Operation

The pumps are switched on/off by the 5V from the Arduino pins going to the each of the TIP120 transistors, which in turn switch the 12V for the individual pumps. Since the Johnny-Five package contains a simple interface for LEDs, I decided to use its switch on/off properties for switching the pumps because it's just a simple digitalWrite(HIGH/LOW). Here's the code for it:

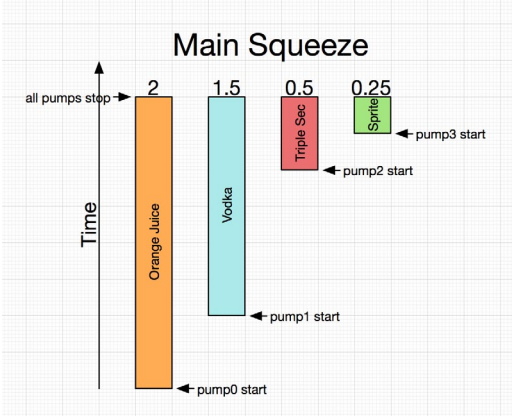
```
> public/javascripts/robot/backend.js
 pump0 = new five.Led(7);
 pump1 = new five.Led(6);
 pump2 = new five.Led(5);
 pump3 = new five.Led(4);
 pump4 = new five.Led(3);
```

The pumpMilliseconds function is used to run a single pump for a number of milliseconds. The usePump function (not shown here) determines which pump to use based on the pump input string.

```
> public/javascripts/robot/backend.js
function pumpMilliseconds(pump, ms) {
 exports.startPump(pump);
  setTimeout(function () {
 exports.stopPump(pump);
 }, ms);
}
exports.startPump = function (pump) {
 console.log("\033[32m[PUMP] Starting " + pump +
"\033[91m");
 var p = exports.usePump(pump);
 p.on();
}

exports.stopPump = function (pump) {
 console.log("\033[32m[PUMP] Stopping " + pump +
"\033[91m");
 var p = exports.usePump(pump);
 p.off();
}
```

Making a Drink

The simplicity of the UI hides much of the complexity behind the actual making of a drink. We want to make a drink that is top-biased in terms of ingredients. That is, all of the ingredients with smaller amounts should be on top so that gravity will cause them to mix into the drink. This adds a little bit of complexity in that we need to also pass a delay amount for each pump, but it is worth it for a drink that is mixed better! Here's a diagram of how the pump timings will work out:



When the user picks a drink and size, the data is stored as a local variable in the Angular $scope.

```
> views/index.jade
div.drinkContainer(ng-repeat="drink in drinks |
orderBy: 'name' | filter: containsIngredients",
ng-click="selectDrink(drink)")

> public/javascripts/controllers/DrinkController.js
$scope.selectDrink = function (drink) {
 $scope.selectedDrink = drink;
 if ($scope.lastSelected) {
 $scope.lastSelected.selectedDrink = '';
```

```
 }

 this.selectedDrink = 'selectedDrink';
 $scope.lastSelected = this;
};
```

The Angular ng-click directive in the jade template file specifies the function to be run in the Angular $scope when the div is clicked. In this case, when the div is clicked, the $scope.selectedDrink variable gets set to the current drink object. When the Make button is pressed, code on frontend.js does two things: 1) it does some visual trickery to turn the Make button into a progress bar, and 2) it does the calculations required to determine how long each of the pumps should fire for based on the ingredients in the drink and the size of the drink selected. So, here's the code for what happens when we tap the Make button:

```
> public/javascripts/robot/frontend.js
$('#make').on('click touch', function () {
 if ($('#make').hasClass('noselection') ===
true) {
  alert('Please select a drink first.');
 return;
 }

 if ($('#make').hasClass('disabled') === true) {
  return;
 }
```

First, we double check to make sure that a drink has been selected first. We can't make anything if there is no selected drink. Additionally, if the robot is already making a drink, the Make button will be disabled and should not make a drink until it is done with the drink it is already making. Next, in the following code, you'll see how we do the visual progress bar for the Make button. We add the "disabled" class to prevent additional drinks from being made until the current one is done, show the hidden #makeProgress div, and then animate it via its margin-left style. At the end of the animation, the anonymous callback function hides the makeProgress bar and removes the "disabled" class. The whole thing is wrapped around a 200ms delay in order for us to get the $scope.pumpTime, which is calculated in the makeDrink function that is explained further down in this section. After this, we call the makeDrink function with the drink's ingredients, the pumps, and the selected drink size ($scope.drinkTime).

```
> public/javascripts/robot/frontend.js (continu-
ing $('#make').on...)
 console.log('Making Drink');
 $('#make').addClass('disabled');
 $('#makeProgress').show();
 setTimeout(function () {
  console.log("Time to Dispense Drink: " +
$scope.pumpTime + "ms");
  $('#makeProgress').animate({
   'margin-left': String($(window).width()) +
'px'
  }, parseInt($scope.pumpTime), 'linear', func-
tion () {
   $('#make').removeClass('disabled');
   $('#makeProgress').hide();
   $('#makeProgress').css('margin-left',
'-10px');
  });
 }, 200);

 // Start dispensing drink
 makeDrink($scope.selectedDrink.ingredients,
$scope.pumps, parseInt($scope.drinkTime));
 });
```

The code below goes through getting the total amount of all of the ingredients, finding the ingredient with the largest amount, and also appending pump labels to the ingredients as a string so that we will be able to determine which pump to use after this data is sent to the backend.

```
> public/javascripts/robot/frontend.js
function makeDrink(ingredients, pumps, drink-
Size) {
 // Check that there are no duplicate pumps
ingredients
 if ($scope.pumpDuplicates > 0) {
  alert("Pump values must be unique");
  return;
 }

// Get largest amount and index of that ingredient
 var largestAmount = 0;
 var amountTotal = 0;
 var largestIndex = 0;
 for (var i in ingredients) {
  amountTotal += Number(ingredients[i].amount);
  if (Number(ingredients[i].amount) > largestA-
mount) {
```

```
   largestAmount = ingredients[i].amount;
   largestIndex = i;
  }

  // Append pump numbers to the ingredients
  for (var j in pumps.ingredients) {
   if (ingredients[i].name === pumps.
ingredients[j].ingredient) {
    ingredients[i].pump = pumps.ingredients[j].
label;
    continue;
   }
  }
 }
```

After all of this, in the code below, you will see that we get the normalization factor, which is the drinkSize divided by the total amount of all drinks. With this normalization factor, we can multiply the largest amount of drink by this value in order to get the total pump time (since pumps will be running in parallel, the total pump time is the pump time of the ingredient with the highest amount). If you recall from above, this is the $scope.pumpTime that we delayed 200ms to get on the front end. After this, we modify the amounts all of the ingredients in the array based on the normalization factor, and add the delay so that we can top-weight the ingredients in the drink. At the end, we use socket.io to pass the ingredients object to the backend.

```
> public/javascripts/robot/frontend.js (continu-
ation of makeDrink function)
 // Normalize
 var normFactor = drinkSize/amountTotal;

 var totalPumpMilliseconds = parseInt(normFactor
* largestAmount);
 $scope.pumpTime = totalPumpMilliseconds;

 // Set the normalized amount and delay for each
ingredient
 ingredients[largestIndex].amount =
parseInt(normFactor * Number(ingredients[largest
Index].amount));
 ingredients[largestIndex].delay = 0;
 for (var i in ingredients) {
  if (i === largestIndex) continue;
  ingredients[i].amount = parseInt(normFactor *
Number(ingredients[i].amount));
  ingredients[i].delay =
ingredients[largestIndex].amount -
ingredients[i].amount;
 }

 socket.emit("Make Drink", ingredients);
}
```

At the backend, app.js catches the "Make Drink" event from the frontend and passes it to the robot portion that handles the actual pumping.

```
> app.js
var robot = require('./public/javascripts/robot/
backend.js');
...
io.sockets.on('connection', function (socket) {
 socket.on("Make Drink", function (ingredients)
{
  robot.pump(ingredients);
 });
> public/javascripts/robot/backend.js
exports.pump = function (ingredients) {
 for (var i in ingredients) {
  (function (i) {
   setTimeout(function () { // Delay implemented
to have a top-biased mix
    pumpMilliseconds(ingredients[i].pump,
ingredients[i].amount);
   }, ingredients[i].delay);
  })(i);
 }
};
```

And that's all there is to it! Thanks for taking the time to read through my post. Hopefully you have learned something; I know I have definitely learned a lot in the process. ■

GitHub link: github.com/ytham/barmixvah

Yu Jiang Tham is a graduate from UCLA in electrical engineering. He loves all of the possibilities 3d printing offers and finds pleasure in creating both hardware and software projects. Yu Jiang creates how-tos on all of his projects at his website *yujiangtham.com* Follow him on Twitter: *@yujiangtham*

Reprinted with permission of the original author.
First appeared in *hn.my/barmixvah* (yujiangtham.com)

HACK ON YOUR SEARCH ENGINE

and help change the future of search

duckduckhack.com

Testing with Jenkins, Ansible, and Docker

By CHRIS LOUKAS

One of our highest priorities at Mist.io is to never break production. Our users depend on it to manage and monitor their servers, and we depend on them, too. At the same time, we need to move fast with development and deliver updates as soon as possible. We want to be able to easily deploy several times per day.

A big part of Mist.io is the web interface so we use tools like Selenium, Splinter and Behave for headless web interface testing. However testing the UI is time-consuming. Having to wait 40 minutes to get a green light before merging each pull request is not very agile.

In this post we'll describe the setup we've used to reduce testing time. It is based on Jenkins, Ansible, and Docker, and its main mission is to automate build steps and run tests in parallel as quickly as possible.

Since execution time is essential for us, we opted to run our CI suite on one of Rackspace's High-Performance Cloud Servers due to their fast performance and short provisioning times. In general, make sure you give your test server enough RAM and a fast disk.

Jenkins

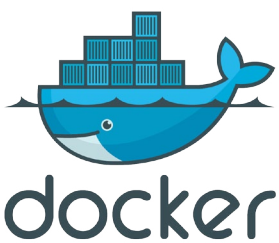
Our first stop for our testing suite is Jenkins. We've used Jenkins in other projects before and we feel comfortable with it since it is mature, widely used, and provides great flexibility through its plugin system. Jenkins has very good Github integration, which is another plus for us. It is quite simple to set it up so that every commit in a branch will trigger the tests.

When our needs started growing, our first thought was to add more Jenkins nodes and have multiple tests running in parallel. But there are many different environments that we want to test every commit against:

- Test the deployment of the app in a clean environment, a fresh build.
- Test the deployment of the app in a staging environment where you want to ensure backwards compatibility.
- Test the web interface against all supported browsers.

All these meant that testing requirements would grow over time, and a testing infrastructure based solely on Jenkins would not do the job fast enough.

Enter Docker



Docker helps you easily create lightweight, portable, self-sufficient containers from any application. It is fast, reliable, and a perfect fit for our needs. The idea is to set up Jenkins so that every pull request to a specific branch (e.g., staging) triggers a job. Jenkins then commands our testing infrastructure to spawn different Docker applications simultaneously and runs the tests in parallel.

This way, we can test any environment. We just have to describe each application the same way we would've done manually. On the plus side, we can use pre-made images from the Docker repository to jumpstart our tests.

For example we could do this:

```
docker pull ubuntu
```

And we will end up with an ubuntu image/application. We can then run the Docker container by typing:

```
docker run -i -t ubuntu /bin/bash
```

And we will be in a fresh, newly created ubuntu environment.

But we don't want to manually run Docker commands after every pull request. What we need are Dockerfiles. Dockerfiles are sets of steps that describe an image/container. We'll use them to build our custom Docker images.

```
Dockerfile commands are fairly simple. For exam-
ple:
FROM ubuntu:latest
MAINTAINER mist.io
RUN echo "deb http://archive.ubuntu.com/ubuntu
precise main universe" > /etc/apt/sources.list
RUN apt-get update
RUN apt-get upgrade -y
RUN apt-get install -y build-essential git
python-dev python-virtualenv
RUN apt-get install -y xterm
RUN apt-get install -y -q x11vnc xvfb
RUN apt-get install -y xfonts-100dpi xfonts-
75dpi xfonts-scalable xfonts-cyrillic
RUN add-apt-repository -y ppa:mozillateam/
firefox-next
RUN apt-get update
RUN apt-get install -y firefox
RUN mkdir MIST
RUN cd MIST && git clone https://github.com/
mistio/mist.io
WORKDIR MIST/mist.io
RUN git pull
RUN cp settings.py.dist settings.py
RUN echo JS_BUILD = True >> settings.py
RUN echo CSS_BUILD = True >> settings.py
RUN echo SSL_VERIFY = True >> settings.py
RUN virtualenv . && ./bin/pip install --upgrade
setuptools
RUN ./bin/python bootstrap.py && ./bin/buildout
-N
ADD ./test_config.py src/mist/io/tests/features/
ADD ./init.sh /
ENTRYPOINT ./init.sh
```

- `FROM` chooses the base image (ubuntu/latest).
- `RUN` runs the following commands. First, we update the system and then we install Xvfb, the latest Firefox, etc., in order to run headless browser steps. Finally, we build the Mist.io app.
- `ADD` adds files from our host machine to the Docker image. In this example, we added the configuration for tests and an init.sh script. The init.sh script could be as simple as this:

  ```
  #!/bin/bash
  cd MIST/mist.io
  git checkout $BRANCH
  ./bin/run_test
  ```

- `ENTRYPOINT` tells Docker to start with ./init.sh script every time we run the image.

To build the Docker image for future reuse:

```
docker build -t mist/iotest /path/to/Dockerfile
```

Now we have a test environment. If there are more tests in other branches that need to be spawned, we can use it for every one of them:

```
docker run -e BRANCH=your_branch mist/iotest
```

If we had multiple test servers, we would have to build every custom image in every test server. Fortunately, Docker lets you have your own private repository of Docker images. You can build your custom image once, say mist/iotest, push it to the repository, and run:

```
docker pull mist:iotest
```

This is a simple test scenario, but the possibilities are endless. For example, in another of our test scenarios we want to spawn a docker application with our monitor service and one with the mist web app.

The problem is that we need every test server configured with Docker and every Docker image available. And we need to automate the procedure to be able to scale our test server infrastructure whenever needed.

Ansible to the rescue

Ansible automates deployment. It is written in Python and installation is relatively simple. It is available through most Linux distro repositories, you can clone it from Github or install it via pip.

Configuring ansible is also easy. All you have to do is group your servers in an ansible_hosts file and use ansible's playbooks and roles to configure them.

For example, this is a simple ansible_hosts file:

```
[testservers]
testserver1 ansible_ssh_host=178.127.33.109
testserver2 ansible_ssh_host=178.253.121.93
testserver3 ansible_ssh_host=114.252.27.128

[testservers:vars]
ansible_ssh_user=mister
ansible_ssh_private_key_file=~/.ssh/testkey
```

We just told Ansible that we have three test servers, grouped as testservers. For each one the user is mister and the ssh key is testkey, as defined in the [testservers:vars] section.

Each test server should have Docker installed and a specified Docker image built and ready for use. To do that, we have to define some playbooks and roles:

```
- name: Install new kernel
  sudo: True
  apt:
    pkg: "{{ item }}"
    state: latest
    update-cache: yes
  with_items:
    - linux-image-generic-lts-raring
    - linux-headers-generic-lts-raring
  register: kernel_result

- name: Reboot instance if kernel has changed
  sudo: True
  command: reboot
  register: reboot_result
  when: "kernel_result|changed"

- name: Wait for instance to come online
  sudo: False
  local_action: wait_for host={{ ansible_ssh_
host }} port=22 state=started
  when: "reboot_result|success"

- name: Add Docker repository key
  sudo: True
  apt_key: url="https://get.docker.io/gpg"

- name: Add Docker repository
  sudo: True
  apt_repository:
    repo: 'deb http://get.docker.io/ubuntu
docker main'
    update_cache: yes

- name: Install Docker
  sudo: True
  apt: pkg=lxc-docker state=present
  notify: "Start Docker"

- name: Make dir for io docker files
  command: mkdir -p docker/iotest

- name: Copy io Dockerfiles
  template:
    src=templates/iotest/Dockerfile.j2
dest=docker/iotest/Dockerfile

- name: Copy io init scripts
  copy: src=templates/iotest/init.sh
dest=docker/iotest/init.sh

- name: Build docker images for io
  sudo: True
  command: docker build -t mist/iotest docker/
iotest
```

After that, we just have to set Ansible to trigger the tests and spawn these Docker applications. All Jenkins has to do is catch the webhook of a new pull request and issue one command:

```
ansible-playbook runtests.yml
```

That's it.

We have set up Jenkins to respond to Github commits and call Ansible to automatically spawn and configure our test servers, and we have optimized our testing speed by using pre-made Docker images. ■

Chris studied Computer Engineering and Informatics at the University of Patras, Greece and is now a Backend Developer, CI/QA Engineer at *Mist.io*

Reprinted with permission of the original author. First appeared in *hn.my/mist* (mist.io)

Build Tools – Make, No More

By HADI HARIRI

I HAD TO UPDATE some CSS on my site over the weekend, which led me to updating some LESS file. The template I use for the site uses Grunt, which forced me to download the entire Internet via npm. And all I wanted to do was set a text-indent to 0.

If you've never seen Grunt, here's what it looks like:

```
module.exports = function(grunt) {

    grunt.initConfig({
        jshint: {
            options: {
                jshintrc: '.jshintrc'
            },
        all: [
            'Gruntfile.js',
            'asselts/js/*.js'
        ]
    }

    ...
```

I find it's very appropriately named, like the ugly little creatures in Halo.

Of course, you don't have to use Grunt. In fact, Grunt is already dead [hn.my/gruntout]. It's all about Gulp nowadays:

```
var gulp = require('gulp');

var coffee = require('gulp-coffee');
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');
var imagemin = require('gulp-imagemin');

var paths = {
  scripts: ['client/js/**/*.coffee', '!client/
external/**/*.coffee'],
  images: 'client/img/**/*'
};

gulp.task('scripts', function() {
  // Minify and copy all JavaScript (except
vendor scripts)
  return gulp.src(paths.scripts)
    .pipe(coffee())
    .pipe(uglify())
    .pipe(concat('all.min.js'))
    .pipe(gulp.dest('build/js'));
});
```

But if you don't like JavaScript, you could probably go with Rake, which is Ruby:

```
task :default => [:test]

task :test do
  ruby "test/unittest.rb"
end
```

If you like Ruby but don't want to install it, and you're on Windows, just use PSake:

```
properties {
  $testMessage = 'Executed Test!'
  $compileMessage = 'Executed Compile!'
  $cleanMessage = 'Executed Clean!'
}

task default -depends Test

task Test -depends Compile, Clean {
  $testMessage
}

task Compile -depends Clean {
  $compileMessage
}

task Clean {
  $cleanMessage
}

task ? -Description "Helper to display task
info" {
       Write-Documentation
}
```

If you don't like Ruby, install Java and use Gradle. It's Groovy and has 65 chapters of documentation as well as a few appendices in case you run out of things to read.

```
buildscript {
    project.ext.kotlinVersion = "0.7.270"

    version = "0.1-SNAPSHOT"

    repositories {
        mavenCentral()
        maven {
            url 'http://oss.sonatype.org/con-
tent/repositories/snapshots'
        }
    }
    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-
gradle-plugin:$ext.kotlinVersion"
    }
}

apply plugin: 'kotlin'
apply plugin: 'maven'
apply plugin: 'maven-publish'

repositories {
    mavenLocal()
    mavenCentral()
    maven {
        url 'http://oss.sonatype.org/content/
repositories/snapshots'
    }
}
```

And if you do like Rake but don't like Ruby, but do like JavaScript, try Jake. [hn.my/jake]

Finally, if you really have nothing better to do, pick one of your favourite XML-based build tools such as Ant, NAnt or MSBuild.

```
<Project xmlns="http://schemas.microsoft.com/
developer/msbuild/2003">
  <ItemGroup>
    <Compile Include="helloworld.cs" />
  </ItemGroup>
  <Target Name="Build">
    <Csc Sources="@(Compile)"/>
  </Target>
</Project>
```

Of course, you could have just used Make to begin with:

```
all: hello

hello: main.o factorial.o hello.o
       g++ main.o factorial.o hello.o -o hello

main.o: main.cpp
       g++ -c main.cpp

factorial.o: factorial.cpp
       g++ -c factorial.cpp

hello.o: hello.cpp
       g++ -c hello.cpp

clean:
       rm -rf *o hello
```

DSL's over Make

If you look at all the examples above, they are nothing but DSLs over Make in your language of choice. In some cases not even that, but just some syntactic sugar. They still focus on the same core concepts:

- Primitives
- Targets
- Tasks

and still force us to transcribe the complexity of a build system in written form.

And with each new Build tool, all we do is just add yet another grammar to learn, another series of tools and plugins to download, configure and install. And for what? Just to have a bit of a nicer syntax? Without for a moment thinking of the legacy that we leave behind with all these new "inventions"? Of course, some languages and platforms might need technology-specific tasks, but is a new build tool and syntax all that necessary?

I think Gradle's tagline is somewhat appropriate to describe all these different build tools: "Automation Evolved". Yes, we've evolved because we've moved from tabs and lines, to XML and then to, well, hmm, tabs, lines and curly braces.

Make, no more.

If you're setting out to make another build tool, think about what benefit you're going to provide over the existing ones. Adding yet another custom DSL and grammar isn't going to really solve any problem.

I feel there is innovation possible in build automation. We have a vast amount of existing knowledge on how software is built and the needs required, we have seen the benefits of using convention over configuration, we have created powerful analysis tools. Combining these things, I feel there is potential to create a build tool that focuses more on discoverability than it does on us having to explicitly declare what we want done.

So please, no more Makes. ■

Hadi Hariri is a Software Developer, currently working at JetBrains. His passions include Web Development and Software Architecture. He has written a few books and have been speaking at conferences for over a decade, on things he's passionate about.

Reprinted with permission of the original author.
First appeared in *hn.my/nomakes* (hadihariri.com)

Better Bash Scripting in 15 Minutes

By ROBERT MUTH

The tips and tricks below originally appeared as one of Google's "Testing on the Toilet" (TOTT) episodes. This is a revised and augmented version.

Safer Scripting

I start every bash script with the following prolog:

```
#!/bin/bash
set -o nounset
set -o errexit
```

This will take care of two very common errors:

1. Referencing undefined variables (which default to "")
2. Ignoring failing commands

The two settings also have shorthands ("-u" and "-e") but the longer versions are more readable.

If a failing command is to be tolerated, use this idiom:

```
if ! <possible failing command> ; then
    echo "failure ignored"
fi
```

Note that some Linux commands have options which, as a side-effect, suppress some failures, e.g.: "`mkdir -p`" and "`rm -f`".

Also note, that the "errexit" mode, while a valuable first line of defense, does not catch all failures, i.e., under certain circumstances failing commands will go undetected.

Functions

Bash lets you define functions which behave like other commands — use them liberally; it will give your bash scripts a much needed boost in readability:

```
ExtractBashComments() {
    egrep "^#"
}
cat myscript.sh | ExtractBashComments | wc
comments=$(ExtractBashComments < myscript.sh)
```

Some more instructive examples:

```
SumLines() {  # iterating over stdin - similar
to awk
    local sum=0
    local line=""
    while read line ; do
        sum=$((${sum} + ${line}))
    done
    echo ${sum}
}
SumLines < data_one_number_per_line.txt
log() {  # classic logger
   local prefix="[$(date +%Y/%m/%d\ %H:%M:%S)]: "
   echo "${prefix} $@" >&2
}
log "INFO" "a message"
```

Try moving all bash code into functions, leaving only global variable/constant definitions and a call to "main" at the top-level.

Variable Annotations

Bash allows for a limited form of variable annotations. The most important ones are:

- `local` (for local variables inside a function)
- `readonly` (for read-only variables)

```
# a useful idiom: DEFAULT_VAL can be overwritten
# with an environment variable of the same name
readonly DEFAULT_VAL=${DEFAULT_VAL:-7}
myfunc() {
   # initialize a local variable with the global
default
   local some_var=${DEFAULT_VAL}
   ...
}
```

Note that it is possible to make a variable read-only that wasn't before:

```
x=5
x=6
readonly x
x=7   # failure
```

Strive to annotate almost all variables in a bash script with either local or read-only.

Favor $() over backticks (`)

Backticks are hard to read and in some fonts can be easily confused with single quotes. `$()` also permits nesting without the quoting headaches.

```
# both commands below print out: A-B-C-D
echo "A-`echo B-\`echo C-\\\`echo D\\\`\``"
echo "A-$(echo B-$(echo C-$(echo D)))"
```

Favor [[]] (double brackets) over []

`[[]]` avoids problems like unexpected pathname expansion, offers some syntactical improvements, and adds new functionality:

Operator	Meaning
`||`	logical or (double brackets only)
`&&`	logical and (double brackets only)
`<`	string comparison (no escaping necessary within double brackets)
`-lt`	numerical comparison
`=`	string matching with globbing
`==`	string matching with globbing (double brackets only, see below)
`=~`	string matching with regular expressions (double brackets only, see below)
`-n`	string is non-empty
`-z`	string is empty
`-eq`	numerical equality
`-ne`	numerical inequality

single bracket

```
[ "${name}" \> "a" -o ${name} \< "m" ]
```

double brackets

```
[[ "${name}" > "a" && "${name}" < "m"  ]]
```

Regular Expressions/Globbing

These new capabilities within double brackets are best illustrated via examples:

```
t="abc123"
[[ "$t" == abc* ]]    # true (globbing)
[[ "$t" == "abc*" ]]  # false (literal matching)
[[ "$t" =~ [abc]+[123]+ ] # true
                          # (regular expression)
[[ "$t" =~ "abc*" ]]  # false (literal matching)
```

Note, that starting with bash version 3.2, the regular or globbing expression must not be quoted. If your expression contains whitespace you can store it in a variable:

```
r="a b+"
[[ "a bbb" =~ $r ]]   # true
```

Globbing-based string matching is also available via the case statement:

```
case $t in
abc*)  <action> ;;
esac
```

String Manipulation

Bash has a number of (underappreciated) ways to manipulate strings.

Basics

```
f="path1/path2/file.ext"
len="${#f}" # = 20 (string length)
# slicing: ${<var>:<start>} or
${<var>:<start>:<length>}
slice1="${f:6}" # = "path2/file.ext"
slice2="${f:6:5}" # = "path2"
slice3="${f: -8}" # = "file.ext"(Note: space
before "-")
pos=6
len=5
slice4="${f:${pos}:${len}}" # = "path2"
```

Substitution (with globbing)

```
f="path1/path2/file.ext"
single_subst="${f/path?/x}" # = "x/path2/file.ext"
global_subst="${f//path?/x}"  # = "x/x/file.ext"
# string splitting
readonly DIR_SEP="/"
array=(${f//${DIR_SEP}/ })
second_dir="${array[1]}"     # = path2
```

Deletion at beginning/end (with globbing)

```
f="path1/path2/file.ext"
# deletion at string beginning
extension="${f#*.}"  # = "ext"
# greedy deletion at string beginning
filename="${f##*/}"   # = "file.ext"
# deletion at string end
dirname="${f%/*}"    # = "path1/path2"
# greedy deletion at end
root="${f%%/*}"      # = "path1"
```

Avoiding Temporary Files

Some commands expect filenames as parameters so straightforward pipelining does not work.

This is where `<()` operator comes in handy as it takes a command and transforms it into something which can be used as a filename:

```
# download and diff two webpages
diff <(wget -O - url1) <(wget -O - url2)
```

Also useful are "here documents," which allow arbitrary multi-line string to be passed in on stdin. The two occurrences of "MARKER" brackets the document. "MARKER" can be any text.

```
# DELIMITER is an arbitrary string
command  << MARKER
...
${var}
$(cmd)
...
MARKER
```

If parameter substitution is undesirable, simply put quotes around the first occurrence of MARKER:

```
command << 'MARKER'
...
no substitution is happening here.
$ (dollar sign) is passed through verbatim.
...
MARKER
```

Built-In Variables

For reference

`$0` name of the script
`$n` positional parameters to script/function
`$$` PID of the script
`$!` PID of the last command executed (and run in the background)
`$?` exit status of the last command (${PIPESTATUS} for pipelined commands)
`$#` number of parameters to script/function
`$@` all parameters to script/function (sees arguments as separate word)
`$*` all parameters to script/function (sees arguments as single word)

Note

`$*` is rarely the right choice

`$@` handles empty parameter list and white-space within parameters correctly

`$@` should usually be quoted like so `"$@"`

Debugging

To perform a syntax check/dry run of your bash script, run:

```
bash -n myscript.sh
```

To produce a trace of every command executed, run:

```
bash -v myscripts.sh
```

To produce a trace of the expanded command, use:

```
bash -x myscript.sh
```

`-v` and `-x` can also be made permanent by adding `set -o verbose` and `set -o xtrace` to the script prolog. This might be useful if the script is run on a remote machine, e.g., a build-bot and you are logging the output for remote inspection.

Signs you should not be using a bash script

- Your script is longer than a few hundred lines of code
- You need data structures beyond simple arrays
- You have a hard time working around quoting issues
- You do a lot of string manipulation
- You do not have much need for invoking other programs or pipelining them
- You worry about performance

Instead consider scripting languages like Python or Ruby.

References

- Advanced Bash-Scripting Guide: *hn.my/abs*
- Bash Reference Manual: *hn.my/bashref* ■

Robert Muth is a software engineer at Google New York. In his spare time he develops Android apps and dances tango, though usually not at the same time.

Reprinted with permission of the original author.
First appeared in *hn.my/bash15* (robertmuth.blogspot.com)

How to Convert a Digital Watch to a Negative Display

By BRIAN GREEN

This project was quite adventurous for me and quite a bit more complicated than some of the other projects I've done with my G-Shock watches. It involves doing some pretty nasty things to the screen of a "naked" G-Shock, so if you're faint-hearted this is probably not the ideal DIY starter project for you. If you're still reading this and desperately wanting to try reversing the display of one of your digital watches, read on!

I'm going to be taking my plain Casio G-Shock DW-5600 and converting the regular display into a negative one with the use of some self-adhesive polarizing film. I bought mine from Polarization.com in Texas. I ordered the thinnest self-adhesive film they had in a relatively small size, part name: "Linear Polarizer w/adhesive PFA."

Ok, on to the project. First let me show you some of the tools you might like to have ready for this.

- Plastic tweezers
- Spring bar removal tool
- Small flat head screwdriver
- Several clean Q-Tips
- A surgical scalpel or sharp modeling knife and fresh blades
- The all-important Husky mini screwdriver (a must-have item)

With all the necessary tools in hand, it's time to start thinking about how to tackle this. I will be using the DW-5600 that I recently stealthed the bezel on. By reversing the display, it should be a pretty fine-looking little watch. The next few steps will be obvious to most of you, but I figured I'd snap some pictures anyway.

Take off the straps so that you can remove the back cover and so that they won't get in the way while you are working on the body of the watch. I like to use my nifty little Bergeon spring bar tool that is designed specifically for this.

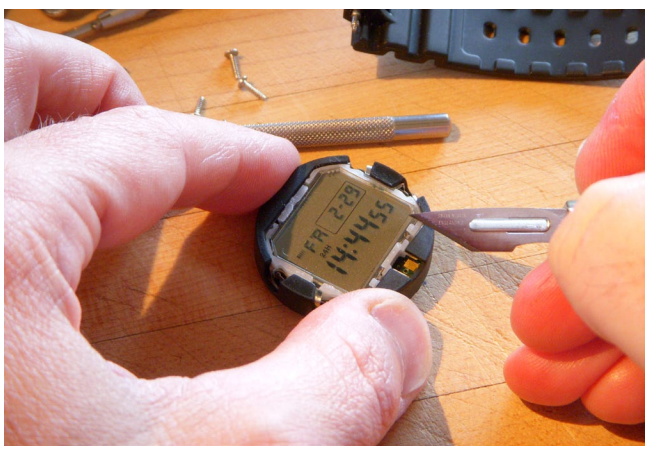
Next, carefully remove the four small screws that hold on the case back. Always make sure to put these somewhere safe and keep them together. This is where the Husky Mini Screwdriver comes in very handy.

Remove the metal case back carefully, trying not to disturb the rubber gasket that creates the watertight seal around the module.

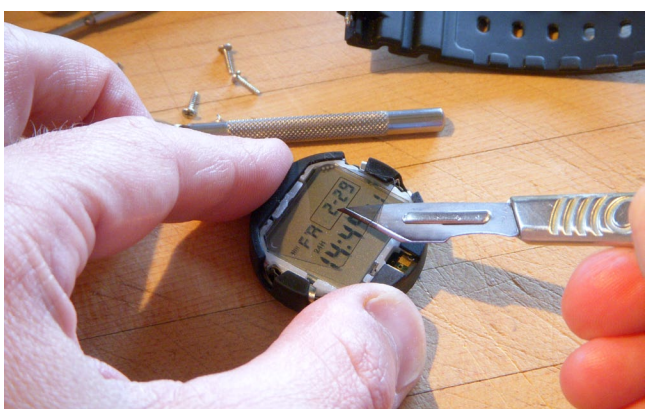
You should see the rubber spacer that covers and adds protection to the inner module. Remove the rubber spacer using the tweezers for extra grip. It can sometimes feel like it is deliberately stuck to the module, but it isn't, it just gets pressed tightly and sticks a bit. It should come off very easily.

You should now be able to lift out the entire module by one of its edges using your tweezers. Mine actually fell right out when I flipped it over. Be patient, there's really nothing holding it in other than the pressure of the buttons against the spring contacts.

I took off on a bit of a tangent here and decided to remove the black outer rubber protector and the metal inner ring casing. I also removed the glass screen from the module and spent the next three hours shouting and cursing at how hard it was to put the darn glass screen back in. I was also extremely annoyed at my stupidity as I discovered that it was not necessary to remove the glass at all (I learn by trial and errors as I go along). I have deliberately omitted the next six or so images that I took of me removing the glass and putting it back in because it is not necessary and very nearly screwed up my display and module!



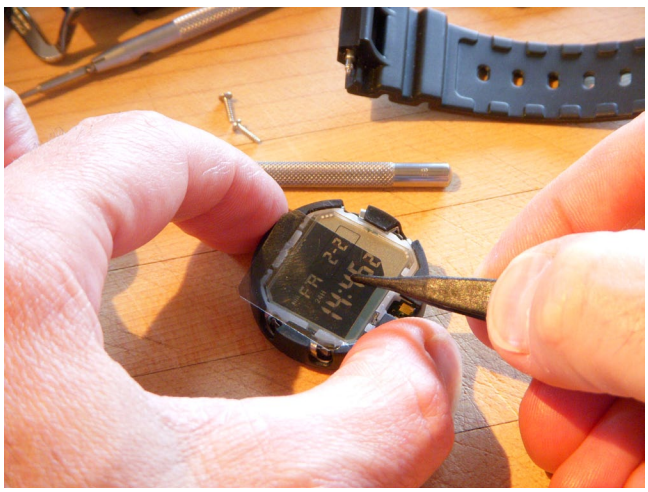
Next remove the polarizing film that is glued to the surface of the glass. The film is slightly smaller than the glass and can be seen easily if you look close up. I am using my scalpel to gently lift up the polarizing film a bit at a time. The trick is to slide the blade between the polarizing film and the glass.



Take your time and work from one edge of the polarizing film across to the other, slowly pushing the blade of your knife under more and more while still moving it from side to side. Eventually you will have the blade under far enough to lift off the polarizing film.

The film is stuck to the glass by a thin layer of tacky glue. It's pretty nasty stuff so be patient and it will come up eventually. Lift off the polarizing film using your plastic tweezers. You can see that the film looks almost transparent while over the display and the digits are only visible on the parts of the display that are covered by the film. It's quite amazing.

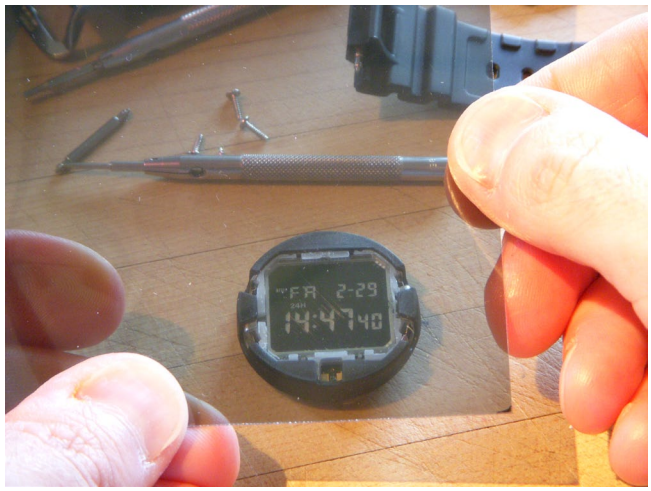
Here's where it gets very cool. Simply turn the polarizing film around 90 degrees and as if by magic the digital display becomes reversed! The polarizing film does not need to be in contact with the glass to work.



At this point I used my Q-Tips and some Goof Off to clean the tacky glue residue from the glass and the old piece of polarizing film. Make sure you get the glass as "squeaky" clean as you can. It took me several Q-Tips and about 15 minutes to get it perfectly clean. I promise you that the time spent getting the glue off as much as possible will be worth it. If there is any glue residue left on the glass it will show up when you stick on the new piece of polarizing film, and you don't want that.

Dry Run With New Polarizing Film

Now let's take a look at the digital module display using the new sheet of polarizing film. Here is the display with the film held in the regular position. The display is shown as normal, and we can see the module is still ticking away quite happily.

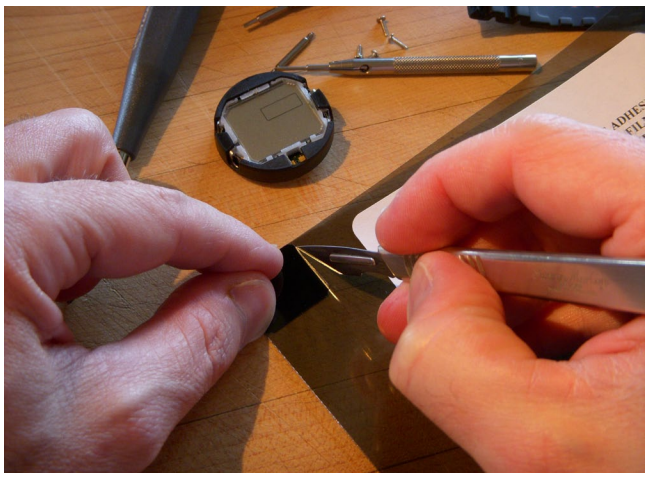


Rotate the polarizing film 90 degrees just like you did with the piece that was removed from the glass and the display is reversed. Excellent, this ensures that the film is going to work. Until this point it was a bit of a gamble on whether or not this particular type of polarizing film would work. Looks good.

Adding the New Polarizing Film

Next you'll need to cut out a piece of the new polarizing film to the exact shape of the original piece. Make sure that you are cutting out the film with it turned in the right direction. Make doubly sure you have the film oriented so that it will make the display look reversed before you place the old piece on top as a cutting guide.

Hint: you can tell when the two pieces are the right way because the original piece that you are using as a cutting template should look completely black. Notice in the picture below how the display is invisible without the polarizing film. Also notice the small box on the display in the upper right corner; this will be disappearing when we reverse the display using this hack. The factory reversed modules manage to reverse the box, too — interesting distinction.



Hold the original piece of film tightly up to the corner of the new sheet and gently cut around it using your sharp knife. Make several slices using medium pressure rather than trying to cut all the way through on the first pass. By making several slices you will avoid slipping and hopefully avoid the loss of any finger tips! Just take your time.

Once you have cut out the new piece of polarizing film, hold it over the display to make sure that it fits and that it will create the desired negative effect. The film used here (details at top) was self-adhesive on one side and had a protective cover on the other. Remove the cover from the self-adhesive side and without touching it carefully place the new piece of polarizing film onto the glass screen.

Use your tweezers for better precision. Gently rub the polarizing film with a soft cloth or clean Q-Tip to make sure it is adequately stuck down. Then use your tweezers again to lift of the protective cover from the front of the film. You should be left with a smudge- and fingerprint-free surface.

The final step is to reassemble the whole thing. Carefully put the whole module back into the watch casing, making sure it is seated down. I find that I nearly always have to use my tiny screwdriver to hold in the metal connectors where the buttons are in order to get a module back in.

Replace the rubber spacer making sure that the protruding metal contacts show through. Then replace the metal case back and four screws. I'm not showing pictures of these steps because most of you know how to do this and if you don't simply read through the steps above that describe how to take the module out.

When the case back is firmly screwed down, flip the whole thing over and admire your handwork: a beautiful, negative display module. Notice how the small box in the upper right corner of the display is no longer visible. This is one difference between the DIY reverse display and the factory fitted version, but I kind of like the minimal look anyway so no great loss for me.

Well that's it. The hardest part of this whole project was biting the bullet on the polarizing film and waiting the couple of days it took for it to arrive. The rest was relatively easy.

I hope you found this a little bit useful and I also hope this encourages a few of you to pop open that old G-Shock and hack a negative display. It took me a little over four hours to do this, but nearly three of those were spent trying to replace the glass display that I shouldn't have removed in the first place. There were also some other distractions along the way.

Happy hacking! ■

Brian Green is the founder of Brian's Backpacking Blog [briangreen.net]. His hacks have been published in MAKE Magazine, featured on *Instructables.com* and won awards. He's not afraid to hack any piece of gear if he thinks he can improve it.

Reprinted with permission of the original author. First appeared in *hn.my/negative* (briangreen.net)

BETTER SENDING, BETTER INSIGHTS

ANALYZE, REACT, ENGAGE

NEW

- REST API
- Parse API
- Real-Time event API
- features coming soon!

mailjet™