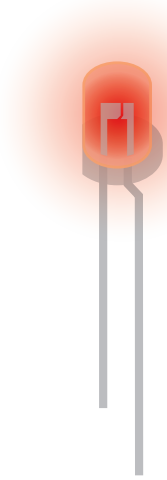


HOW TO LIGHT THE ENTERPRISE

by Jennifer Petkus

Introduction

I have been in love with the Enterprise since the original *Star Trek* show aired in the 1960s. The ship is an improbable melding of grace and power, implying an advanced technology is responsible for making such a delicate looking ship possible. When the Enterprise refit was introduced in the first Star Trek movie, I was even more in love. It is the epitome of "Starfleet clean," ships so beautiful they seem ethereal—a far cry from the world weary, grungy look of Star Wars. Starfleet vessels are largely free of greeblies—details added to break up surface areas. Instead, the Enterprise refit was detailed through an intricate painting process that made it shimmer when seem from certain angles. There were both subtle variations and a larger, barely discernible pattern called "azteking," that again implied a technology beyond our ken.

The refit was also bedight with a multitude of lighting effects, including collision strobes; navigation/formation lights; deflector dish and warp nacelle glows; RCS thrusters; and spotlights that shone on the ship's registry markings. I'm a little embarrassed to admit—but not ashamed—that I tear up when I see the spacedock startup sequence in the first two movies, when the spotlights turn on in sequence to show the grand, gray lady waking up.

I've been trying to build the Polar Lights 1/350th scale USS Enterprise refit for more than a decade now.[1] It's one of those very long-term projects that has languished principally because I've been unsure of how to light it. At first, I tried making a lighting circuit based on simple integrated circuits, like

1 Actually my efforts at lighting an Enterprise go back to the AMT 1/537th scale kit and Paul M. Newitt's *Starfleet Assembly Manuals*

the 555 timer, but there was a limit to what my extremely limited knowledge of electronics could do. Then a friend gave me an Arduino Uno, essentially a computer on a single chip, that could, if I learned to program it, power the Enterprise.

Still, the project languished because although the programming wasn't that difficult, it would still require a lot of circuitry to make it glow. Then I found a series of tutorials on the YouTube channel Ostrich Longneck that explained how to fire phasers, light the deflector dish and go to warp. But it wasn't until 2020 and an excess of free time allowed me to design my own lighting circuit.

Starship lighting circuit

Eventually I ended up with a circuit based on two microprocessors, the ATtiny85 and ATMega328, plus a number of other integrated circuits, transistors, resistors and capacitors. I've managed to get just about every feature I wanted, including some I didn't even know I wanted. These features include:

- Collision strobes
- Navigation/formation lights
- Deflector fade up/fade down
- Fade up/down blue warp nacelle glow
- Impulse engine fade up/down
- Photon torpedoes
- Phasers
- RCS thrusters
- Spotlights
- Sound effects

The lighting effects are mostly based on the first two Star Trek movies, *The Motion Picture* and *Wrath of Khan*. I've made certain decisions about these effects that don't actually match every detail from these movies or subsequent movies featuring the refit or the Enterprise-A. For instance, I've decided:

- When at impulse, the main deflector and impulse deflection crystal glow amber. The magnatomic amplification crystal and the warp nacelles do not glow.
- When at warp, the main deflector and impulse deflection crystal glows blue. The magnatomic amplification crystal and the warp nacelles glow blue.
- When at impulse, the impulse engine glows red; it does not glow when at warp
- The RCS thrusters only fire when at impulse, not at warp

Some of these decisions are the result of my trying to observe the rules of how starships operate, even if the movies violate them, but frankly most of them, are the consequences of the limitations of my programming skills or the limitations of the chips that power the starship.

In that first category, I believe that the warp nacelles shouldn't glow when at impulse, and you see that, for instance, during the Mutara Nebula battle in *Wrath of Khan*, when the Enterprise and Reliant are playing pin the tail on the starship. You'll see the Enterprise passing over the Reliant, with the warp nacelles dark, because the ship is at impulse or perhaps even just using the RCS thrusters. But in other movies you'll see the nacelles glow when at impulse. I can imagine reasons why the nacelles might glow when at impulse, but not as a

general policy.

Of course, in the same nebula scene, you'll notice that the impulse deflection crystal glows blue, when I think it should glow amber, reflecting the Enterprise is at impulse. But, of course, in the same battle, you'll see the deflector glowing blue.

Here's where my pedantic need to impose rules on what is just entertainment collides and colludes with the limitations of my skills and the limitations of my materials. We'll go into this in detail later, but for now I'll explain that two pins on the ATtiny chip are responsible for lighting the amber and blue LEDs to represent impulse and warp. The same impulse pin is responsible for lighting the LEDs for the deflector, crystal and impulse engine. The problem is that according to that famous spacedock startup sequence, the deflector glows while in spacedock but the impulse engine doesn't glow until the ship leaves spacedock.

Unfortunately, there's no way to accomplish this with the circuit as designed. Every pin on the on the ATtiny and ATMega is already assigned, so there's no free pin to separately light the impulse engine. If I really wanted to, however, I could eliminate one of the two phaser pins of the ATMega—resulting in always lighting the same phaser emitter—and assign it to lighting the impulse engine. That's unappealing because now the impulse lighting effect is handled by the ATtiny and ATMega, and it would be difficult to coordinate.

I could also do some ugly, low-level reprogramming to turn the reset pin of the ATtiny into a digital input/output pin, but if you make a mistake, you might end up with a bricked chip. Admittedly, the chip is pretty cheap and there are ways to fix the chip, but I don't think most people would want to attempt this. This is one of those cases where I'm content to have the impulse engine start while in spacedock.

Should you do this?

Before you decide to build your own starship lighting circuit, I should mention there are several companies that sell pre-made circuits that will do just about everything you might desire. Although these circuits aren't inexpensive—you might spend $250—the cost of buying all the components you need to build your own add up. This includes a supply of assorted resistors and capacitors, wiring, breadboards, perfboards, soldering iron, wire strippers and assorted tools like multimeters, tweezers, pliers and flush cutters. You could easily spend several times over that $250.

Of course, if you already have a lot of this equipment and components, then the cost shifts in your favor. For instance, you'll only need one 16Mhz clock crystal for this project, but I bought a package of 20 for $7. I have enough components to light three starships. I was also able to add specific features that I wanted, so my lighting board is unique.

I will warn you that you're in store for a lot of frustration, interspersed with occasional cries of success. Several times you'll be tempted to give up, but I hope this guide will help. I would be delighted if your design differs from and exceeds mine. In the next chapter, you'll find out what you'll need to know if you attempt this.

Development Work Flow

My development work flow should be different, but still similar to yours. Hopefully I've done most of the ugly stuff for you, but you still might have to do some grunt work if you decided to add different features to your board.

I realize that this may seem like gibberish now, but come back to it after you've read the rest of this book.

I started by writing sketches for the Arduino Uno connected to a breadboard. I wrote the ATtiny sketch first (strobes, nav lights and deflector) because it's relatively simple. It worked perfectly the first time, but then I had to upload the sketch to the ATtiny and nothing worked. After a day of scratching my head, I realized that I had forgot to change the Uno (or ATMega) pin assignments to the corresponding ATtiny assignments.

Then I wrote the spotlights, weapons, RCS thrusters and sound sketch that would go on the ATMega. Testing it on the Uno was a slow process as I discovered the mistakes I'd made, but it was steady progress. Uploading the sketch to the ATMega, however, was very painful because I'd forgotten the process that you'll find in the chapter Programming the Microprocessor. Even after I successfully uploaded the sketch, however, I still had problems getting all the components working. Getting the Adafruit FX Sound Board working was especially problematic until I told the board to reset in the setup block.

Finally, I had both sketches working and uploaded to the appropriate microprocessors. The next step was getting both microprocessors and the other ICs and components on a single breadboard, with the onboard power supply. You can read about these problems in the More About … chapter.

Of course that wasn't the final, final step. I had to transfer everything to the solderable perfboard and of course I made mistakes. I'm sure you will make mistakes, which is why you'll need the de-soldering tools mentioned in the What You'll Need chapter.

If you decide not to add any features or make any changes to the sketches, you might be able to skip the testing on the Uno and proceed to uploading the sketches to the microprocessors. (You'll still want to test compile the sketches in the Arduino IDE.) But debugging sketches on the microprocessors is difficult, so if there are any problems, you'll probably want to test using the Uno.

I realize this is daunting, but hey, this is easy compared to painting the Enterprise.

What You'll Need to Know

This book is not intended to teach you everything you need to know before you start building your starship lighting circuit. I'm afraid you'll either need to already have certain skills or acquire them before you start. It's outside the scope of this book to teach you these skills. So, what are we talking about:

Soldering

Although you could build the lighting circuit on breadboards, which requires no wiring, they're not suitable for long-term use. A breadboard circuit has a lot of noise because the connections are tenuous—just a friction fit really. Also, components and jumper wires can easily come loose, and breadboards are bulky.

So, you will need to know how to solder components to a prototyping board, and the only way to learn is to practice. Fortunately, if you just type "how to solder" in YouTube's search bar, you'll find numerous tutorials. Watch of few of these and you will learn the basics. After that, it's just practice.

I do feel like I should offer some pearls of wisdom, however. There are a few things I've done or learned that have greatly improved my soldering skills.

- I bought a really nice soldering iron with a digital temperature control readout
- I have vises and clamps to hold my work steady
- I've learned to compulsively clean and re-tin the soldering tip
- I've learned to apply solder to the part(s) I'm trying to connect, not the soldering tip
- Pre-tin any wires and any pads to which you're soldering

We'll address the soldering iron and the vises and clamps in the What You'll Need chapter. Let's look more closely at the last two tips.

Clean and re-tin

It's extremely important to constantly clean and re-tin the soldering tip, otherwise you'll need to constantly buy new tips. Solder tips oxidize (corrode) because of the high temperatures involved, but a clean tip protected by a coating of solder will last a long time.

If I'm soldering a resistor to a perfboard, for instance, I'll first drag the tip across a wet sponge, then through a brass scouring pad, add fresh solder—just a little—to the tip and immediately solder one leg of the resistor to the board.

Then I drag the tip across the wet sponge again, then through the scouring pad, re-tin and immediately solder the remaining leg. After several soldering several components, I'll instead clean the tip with the sponge and scouring pad and then put the tip in a small jar of tip tinner, run it through the scouring pad and then add a little solder.

Whenever I have to put the soldering iron back in its cradle, I clean and re-tin the tip with solder, even if I'm pausing for a few seconds. Of course I sometimes to forget, but I mostly remember to clean and re-tin after each use.

Solder the part, not the tip

Apply the soldering tip to the thing you're connecting to, like the circular pads on a perfboard, and the thing you're connecting, like the leg of the resistor. Touch the solder to the pad, not to the tip of the soldering iron. The idea is not to add a lot of solder to the iron and hope it leaks onto the thing you want to attach. The idea is to heat the two parts and apply solder to the parts, allowing capillary action to draw the solder into the spaces between the parts.

Pre-tin wires and pads

You'll avoid overheating integrated circuits and other components like LEDs if you pre-tin any wires or soldering pads. If I'm soldering a resistor to an LED, for instance, I'll first tin the legs or leads of the LED and the resistor before attempting to solder them together. You'll find pre-tinned leads join much more quickly, preventing overheating of components.

Use flux when needed

You can usually guess when a soldering connection will be difficult, like joining a small thing to a large thing, two large things or two tiny things. A dab of flux will make the job easier. However, remember to …

Clean up after soldering

Soldering rosin and flux can leave a lot of residue that could either corrode or cause unwanted short circuits. You should clean the residue with flux cleaner, or make your own with a 50/50 mix of isopropyl alcohol and acetone

Programming

This book also isn't intended to teach you basic programming concepts. It's up to you to learn the basic concepts of variables, loops, if statements and programming blocks. Fortu-

nately, the two sketches that make up our starship lighting circuit are very simple, and any previous programming experience is all that's required to become familiar with C++.

C++ and Arduino programming is new to me, and I was constantly googling the syntax of basic functions. When I didn't find the answers, I found repeatedly slamming my head against the desk does work … after a day or so. In one case, I went to the Adafruit support board and posted a question and got no replies. So, I started replying to myself, suggesting what might be wrong and eventually answered my own question. It did take two days, though.

Electronics

Again, I can't teach you electronics. I barely understand it myself, and frankly, you don't really need to know a lot to make the circuit. If you follow the diagrams precisely, you'll have no problems (assuming I've made no mistakes).

But following the diagrams can be extremely difficult. I spent two days trying to figure out why the final circuit wasn't working. Luckily, I have the habit that if a circuit I'm building isn't working as expected, I immediately kill the power. Thank goodness, because what I'd done is connect one of the ATMega pins to power, not ground, and had created a short circuit. The circuit was drawing more and more power until the resettable fuse kicked in.

I didn't realize this immediately, however, and tried other remedies. I'd re-connect the circuit to power, see it wasn't working and kill the power. I spent hours looking at the circuit wondering what I'd done wrong. It wasn't until I touched the resettable fuse and found it hot that I realized I must have a short circuit, but I still couldn't see my mistake. It wasn't until I removed the ATMega and saw that the ATtiny was working that I realized what I'd done wrong. Fortunately the fuse did exactly what it was supposed to do.

One great way to learn both Arduino programming and electronics is at TinkerCad. This is a free website that lets you prototype circuits virtually. You can even program a virtual Arduino and hook it up to a virtual breadboard containing virtual components. The advantages are you don't need to buy an Arduino and you'll always have the right value resistor or capacitor. It's much easier to wire a virtual breadboard and if you make a mistake, you're only burning out a virtual component.

There are disadvantages, of course. TinkerCad supplies a lot of components, but not, for instance the ULN2803A Darlington array or the Adafruit Sound Board, so you can't test these out. Nor does it allow you to load additional libraries, like LedFlasher[1] or arduino-timer; you're limited to the standard "built-in" libraries.

Nevertheless, I was able to virtually test parts of my circuits and sketches and then put it all together in the real world.

1 I was able to copy out the code of the LedFlasher and paste it into the top of my sketch. Basically, I was creating a class within the sketch instead of using an #include to load it.

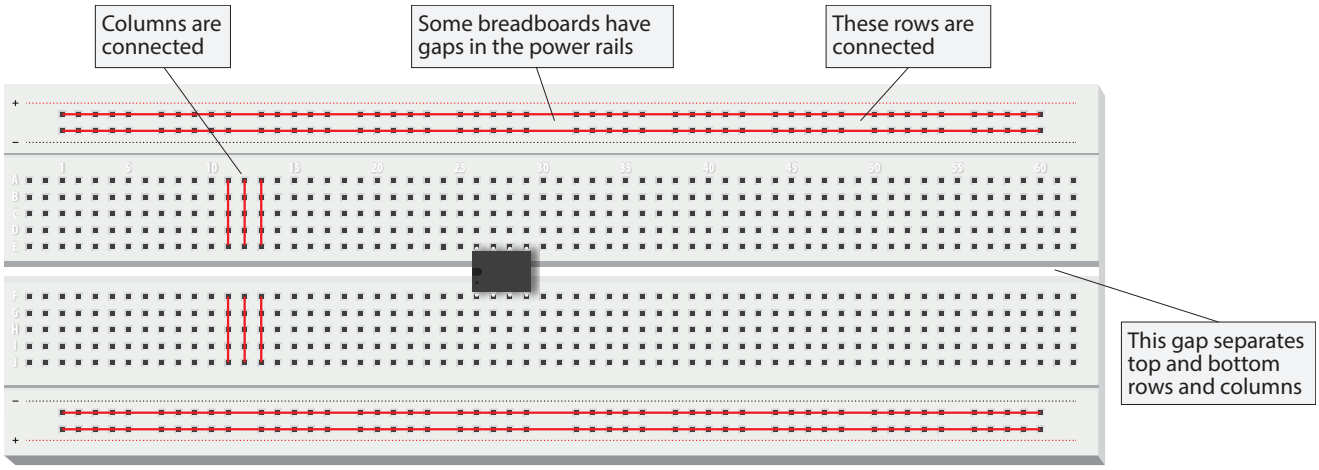
Debugging

You're going to make a mistake somewhere, either with the electronic components or the sketches. To figure out the problem, use the 50/50 method. In programming, you do it like this. Somewhere in your sketch you've made a mistake, but you don't know where. Either the sketch won't compile or it doesn't do anything. The error message the Arduino IDE returns is cryptic and unhelpful. Or worse yet, you're uploading a sketch directly to the ATtiny or ATMega and there are no error messages.

To find the problem, comment out half your code. Of course, you'll also need to comment out any calls that refer to the half of the code you commented out. Upload the sketch again—it still won't do anything, but if it suddenly compiles, then whatever programming *faux pas* you've committed is somewhere in the code you commented out. Then uncomment half of that code and if it still compiles or at least some of the lights are blinking, then the problem is in the remaining uncommented code, or vice versa.

You can debug the circuit the same way. Strip away half the circuit and if it works, the problem was in the half you removed. You might move the ATtiny and everything connected to it—transistors, MOSFET, resistors, to a separate breadboard, give it power and see if it works.

Another trick is to recreate the problem in TinkerCad, if possible. That might not be possible if it involves a component or library unavailable in TinkerCad, but you can probably still recreate some part of the circuit. Perhaps you inserted a transistor into the breadboard backwards. That mistake will be a lot more obvious in TinkerCad because it labels the pins of the transistor, and suddenly you'll realize you're tying the emitter to ground instead of the collector.



What You'll Need

At a minimum, you'll need these tools:

- Breadboards, on which you'll test circuits
- Jumper wires to connect the components on the breadboard
- Solderable perfboards, on which you'll mount your finished circuits
- Wiring: I bought an assortment of 26- and 30-gauge wiring in six colors
- Soldering iron and solder
- Wire strippers capable of stripping the wiring mentioned above
- Tweezer and pliers

You might also want:

- Chip removal and insertion tools: it's very easy to bend the legs of ICs as you insert or remove them from a breadboard. These tools are cheap and very handy.
- Multimeter: I use mine primarily to check continuity, voltage and resistor values—occasionally for amperage
- Third hand tool and/or vise for holding wiring and components while soldering
- Crimping tool for making Dupont and JST connectors

I have a dedicated modeling desk with several work lights to see what I'm doing

- Resistors and capacitors are best bought as assortments, with individual purchases required if you lack specific values

Breadboard

You use breadboards to build and test your circuits. You can plug components into the breadboard without soldering and connect them with jumper wires that again simply plug into the holes of the breadboard. In the full-size breadboard shown above, you can see that the columns are connected, so each hole in a column has an electrical connection to the hole below. The holes are spaced 0.1" (2.54 mm) apart, which more or less corresponds to

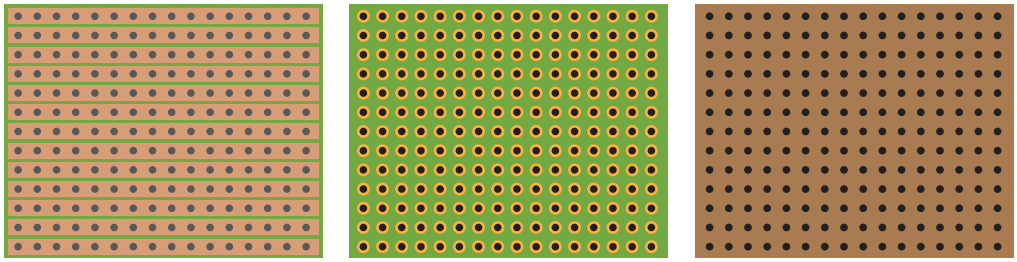
most components, such as the spacing of the legs of an IC.

The top and bottom rows are connected and are used as power rails, so a positive voltage or ground connected to any hole in the row is available to all the holes. In some breadboards, however, there's a break in the middle, which allows you to have a different voltage like 3.3V on the left and 5V on the right. Jumper wires can bridge the break to make a continuous voltage. There's another break in the middle of the board that separates the columns in the top half from those in the bottom, and also breaks the power rails at the top from those at the bottom. Often a jumper is used to connect the power rails top and bottom.

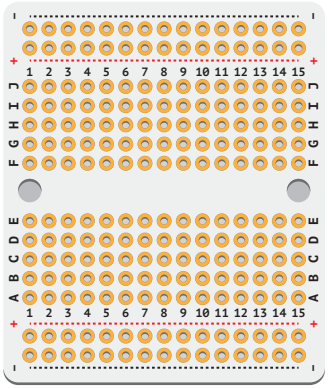
Integrated circuits—or chips—are inserted into the breadboard horizontally, straddling the gap through the middle. Jumpers plugged into the holes above and below the chip then connect directly to the pins of the chip.

Breadboards are used for prototyping—seeing if the thing works—and are not intended for long term use. Just plugging a wire into a hole is a tenuous connection and jumper wires often break free, confounding you if you didn't notice.

Solderable prototyping or perfboard



For permanent use, you will want to solder components onto a solderable perfboard (also called a solderable prototyping board). A perfboard has the same spacing as a breadboard, and there's a lot of different layouts. In the picture above, the first example shows connected rows. The second example has individual pads and the last has holes but no solderable pads.



I really like the Adafruit Perma-Proto Breadboard PCB, which comes in quarter-, half- and full-width sizes. These boards are clearly labeled and have the same layout as a breadboard, making it easy to transfer a circuit designed on a breadboard.

Soldering iron

Get a decent soldering iron with a temperature selector. I have a Hakko FX888D that costs about $100, but you could find a good Weller variable temperature soldering station for under $40. A soldering station usually includes a holder and a sponge for tip cleaning. The Hakko has an additional coiled brass tip cleaner. I can precisely set the Hakko for 500° F, which adequately melts the solder but doesn't destroy the tip.

By the way, solder is usually composed of lead, tin and rosin. You'll want good ventilation to remove the rosin fumes and you should wash your hands after handling solder.

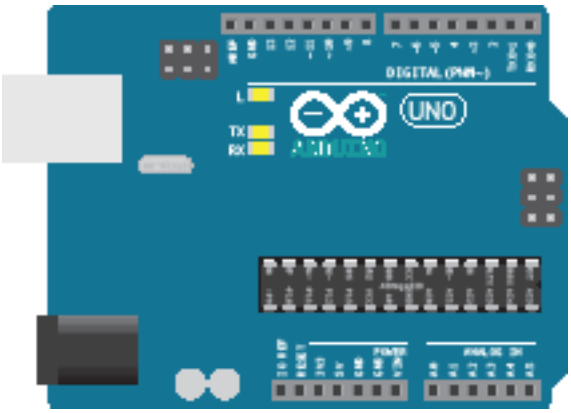
You'll probably also want a solder remover tool and/or de-soldering wick, for when you make a mistake and need to remove a component.

Another good investment is a third-hand tool and/or vise, but most modelers already have these. And finally, locking tweezers or heat-sink clamps would be useful to prevent overheating components, but again, most modelers probably have these.

Dupont/JST crimper

You can soldering connecting wires directly to the perfboard, but there are many places where it would be nice to have a connector that can be easily attached/removed. I've bought a crimping tool to make Dupont header pins and JST-XH connectors. The crimping too is not a must have, but I prefer neat and tidy wiring and hope to avoid the nightmare of spaghetti wires into the saucer section.

Arduino



You will need some flavor of an Arduino, and I'd recommend getting an Arduino Uno because it uses the same chip you will use on the starship lighting circuit. It comes with header pins (actually the female half) that make it easy to run a jumper wire from the Uno to a breadboard. You could probably buy a cheaper Arduino clone, but at $23, I think you'll be better off getting a genuine Arduino. You could probably also get an Arduino Nano, but you'll need to plug it into a breadboard or use a shield to make easy connections, at which point you'll basically have an Arduino Uno.

Incidentally, there is an Arduino Mega 2560 that has a lot more input/output pins than the Uno. You probably don't want to use this because it uses a different processor than the ATMega328. I mean you could, but you'd have to keep changing the pin numbers when testing on the Arduino Mega 2560 and when uploading to the ATMega328. You might also be tempted to use those extra pins, only to find out they're not available on the ATMega328. Also, I think the ATMega 2560 chip is only available in a surface mount design, making it nearly impossible to use with a perfboard.

Bench power supply

You may wonder why you need a bench power supply, considering that we're making a 5V supply right on the main board. But it's very nice to have a variable supply with a lot of amps to power your circuits without having to create a power supply from scratch each

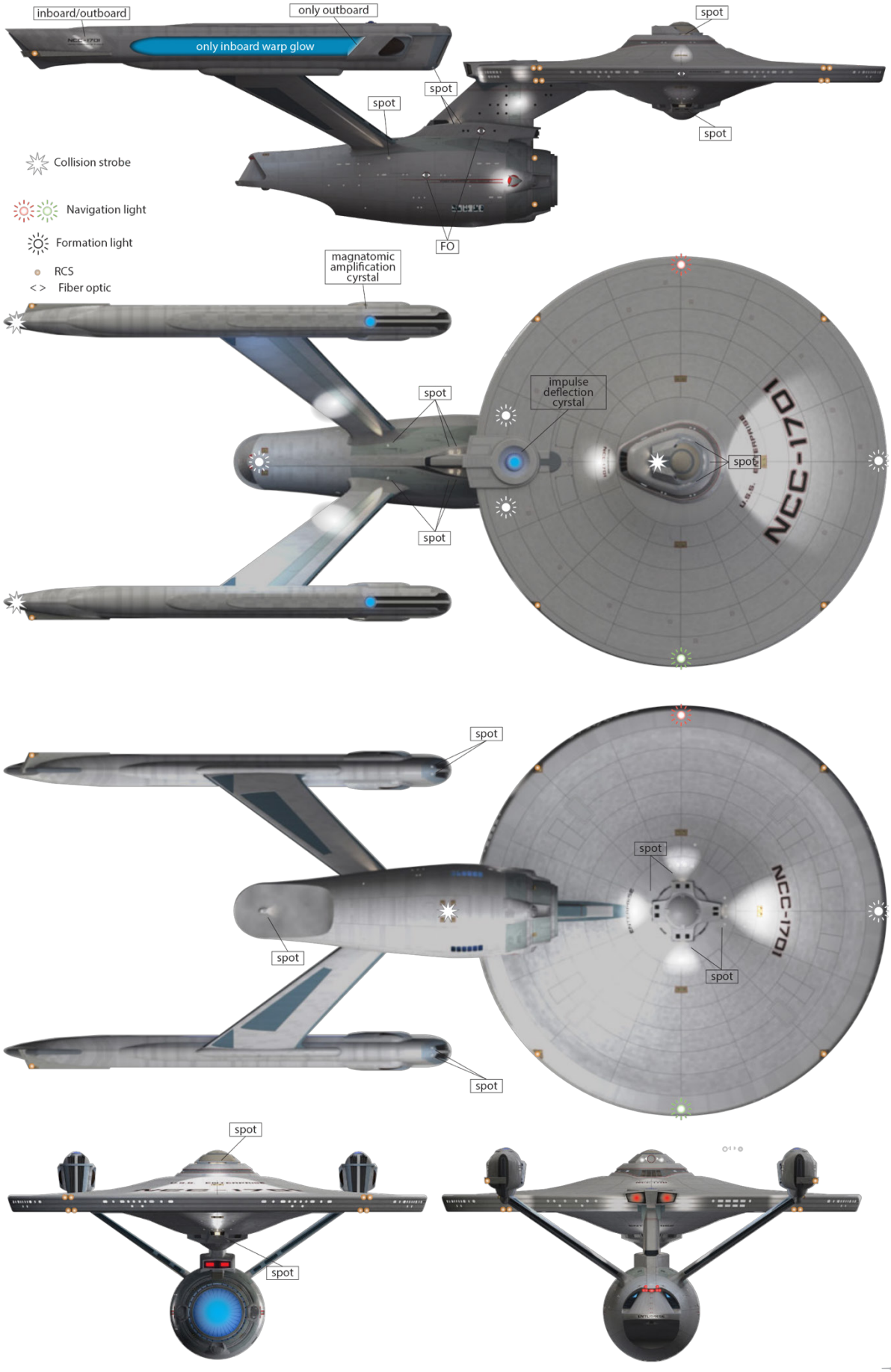
time.

A bench power supply is a luxury item to be sure, but it certainly comes in handy if you make a lot of circuits. I've made a little variable supply using an LM317 voltage regulator and there are a ton of tutorials that cover this. You can buy a bench power supply for $50–75. Another popular option is to cannibalize an old ATX power supply, which will have steady 3.3, 5 and 12V output. Just trim away the wires you don't need. You'll need to add some plugs and a case, but it's very simple. If you want to get more ambitious, add a variable supply using an LM317 and a digital readout that shows voltage and amperage. The latter is extremely useful in determining what size power adapter you'll need for the finished model. If you see that your circuit is drawing 1.4 amps, then you'll know you need a 2 amp power supply. Note: Give yourself room on power supplies. Using a 1.5 amp supply would probably run hot.

Multimeter

Of course, if you have a multimeter, that will also show you how many amps your circuit is drawing. You don't need a very fancy multimeter for this project. You'll mostly be checking continuity—whether there's a solid connection between components. I occasionally use a multimeter to double-check the value of a resistor if I'm too lazy to look it up. And as mentioned, I also use it to determine the current draw of a circuit.

In the next chapter, we'll take a closer look at the components you'll be using in the final circuit.


inboard/outboard
only outboard
only inboard warp glow
spot
spot
spot
spot
FO
Collision strobe
Navigation light
Formation light
RCS
< > Fiber optic
magnatomic amplification cyrstal
spot
spot
impulse deflection cyrstal
spot
NCC-1701
spot
spot
spot
spot
spot
NCC-1701
spot
spot
spot

Lights and sound

There are a mind-numbing number of lights on the Enterprise refit and, of course, it's up to you whether to include every light and lighting effect. If all you want to do is get the collision strobes and navigation lights flashing that's perfectly understandable. My goal was more ambitious and if you're reading this, I suspect yours as well.

Spotlights

One of the most maddening lighting effects on the refit are the spotlights. I'm sure anyone who is attempting to light a refit are already aware of the problems, but to summarize, in some places it's very difficult to make a spotlight shine on that spot or even physically impossible. Most of the spotlight effects on the filming model were accomplished with external lighting. How you achieve this impossible lighting is up to you, but most people resort to the Raytheon effect. Just Google it if you're unfamiliar.

I count 25 spotlights or spotlight lighting effects on the model, but if you following the spacedock lighting sequence, you can group these lights into seven locations. There's an excellent lighting reference on YouTube that shows the exact time the spotlights activate and in what order.

Nacelle pylons

There are spotlights shining on the outboard and inboard sides of the nacelle pylons. Unfortunately, the angle of the spots could not possibly produce the round glow effect seen on the pylons.

Nacelle front

There are two spotlights at the front of the nacelle. The outboard spot shines on the edge of the saucer near the RCS thruster quadrant. The inboard spot probably shines just to the outside of the impulse engines.

Saucer pylon

There are two spotlights on either side of the base of the pylon that connects the engineering hull with the saucer. One is certainly responsible for the large oval lighting effect seen on the side of the pylon, but the angle of the spotlight to the pylon could not possibly produce that effect. The other spotlight seems to shine up to the impulse engines, however that conflicts with the inboard spot at the front of the nacelle, which also seems to shine on the impulse engines.

Nacelle rear

There are two lighting effects both inboard and outboard on the registry at the rear of the nacelle. There's no logical source for this lighting except for something like electro-luminescent paint?

Saucer above

There are three clearly identifiable spotlights at the base of the bridge (A deck) that shine port, starboard and forward. It's difficult to get these lights to actually shine on the saucer as shone, but it is just possible by slightly increasing the height of A deck. To reproduce what's seen in the movie, however, requires the Raytheon effect.

There is no visible source, however, for the spotlight that shines on the registry aft of the VIP lounge and will require the Raytheon technique to reproduce.

Saucer below

Fortunately, the source of the four spotlights that emanate from the planetary array on the underside of the saucer are obvious, however it is difficult to angle the lights to produce the movie lighting.

Engineering hull

There are spotlight effects on either side of the engineering hull shining on the Starfleet pennant, but there's no logical light source. There's another spotlight found on the underside of the engineering hull emanating from the bulge that's the cargo bay tractor beam emitter. It shines on the large cargo hatch. Finally, there's a spotlight shining on the name of the ship on the lip of the main shuttle bay with no logical light source.

Startup sequence

Based on the lighting reference mentioned earlier, you can see that in order:

1. The spotlights at the base of the nacelle pylons light;
2. then the spotlights at the front of the nacelles;
3. next the spotlights at the base of the saucer pylon;
4. the lights on the registry at the rear of the nacelles;
5. the lower saucer spotlights;
6. the upper saucer spotlights;
7. and finally, the spotlights shine on the pennants on the engineering hull.

The RCS thrusters glow when at step four when the registry is lit on the nacelles and I assume all the thrusters throughout the ship glow. The deflector dish also glows at step seven and I've decided that's when the collision strobes and navigation lights should begin flashing. I've also decided that's when the tractor beam spotlight activates.

Apart from the seeming impossibility of some of these spotlights, the biggest puzzle is the source of the lighting of the impulse engines and the Starfleet pennant on the engineering hull. In the YouTube lighting reference referred to earlier, it's clear that both the outboard and inboard spotlights at the front of the nacelles light at the same time. Is the inboard spotlight shining on the impulse engines?

Two seconds later, the spotlights at the base of the saucer pylon light. The forward spot seems to light the pylon while the rear spot seems to shine on the impulse engines.

I think it makes more sense that the inboard spot on the nacelle shines on the engineering hull and the rear saucer pylon spot lights the impulse engines.

Which, of course, conflicts with the lighting of the engineering hull pennant at 11 seconds. The only way to reconcile this is to either assign the pennant lighting to luminescent paint or decide the inboard spot does not light at the same time as the outboard spot.

That decision is up to you. If you decide to light the two forward nacelle spots at different times, you'll need two wires going to the nacelle for that purpose.

RCS thrusters

There are far more RCS thrusters than the eight pins on the 74HC595 chip that is used to light them. However, if we group the thrusters, we can come up with eight. The thrusters on the saucer, for instance, are already grouped into four quadrants, even if each quadrant has separate upper, lower and side thrusters. We can do the same for the thrusters at the ends of the nacelles and just treat upper, lower and side thrusters as a unit. That gets us up to six.

There are four thruster locations that ring the deflector dish, but we can group these into port and starboard thrusters, for a total of eight groups.

Now, as shown in the startup sequence, the thrusters are always faintly lit. It's up to you

how you want to handle this, but I've decided that where possible I'll run fiber optic to each thruster from window lighting and where necessary I'll run an always on LED. What is not possible, however, is to match the lighting of the RCS thrusters in step four. They will appear lit the moment power is supplied to the circuit.

Window lighting

Window lighting is provided by LED strips placed throughout the ship. Unfortunately, there's no way to control the intensity of the LED strips because we've run out of pins on our two processors. We would need a PWM pin for this purpose, and that would require reassigning a pin (perhaps the second phaser pin) and some shuffling to make a PWM pin available.

There are a few workarounds, however. If your power supply adapter (the wall wart) can be set to different voltages, you can select 9V instead of 12V. It will be a little bit dimmer, but probably not much. You could buy a dimmer for the LED strip, but it might look out of place, especially if the hardware is obtrusive. (Dimmers with a remote control would be preferable.)

Or, if you have a spare 555 timer, you could make a PWM dimmer. Because there's no room for it on our board, you'll need a separate perfboard. You'll end up with a potentiometer and a MOSFET connected to a 555 to dim the LED strips.

Of course, at $1.30 for a 555, you might consider just buying another ATtiny85 for $1.64 (these are prices for individual components at Mouser.com). You'd only need two of its five pins, leaving you three more pins to add additional functions. You could even add a sensor to adjust the intensity of the LED strips depending on whether it's day or night. Or, you can just live without the ability to dim the LED strips.

The 555 solution, however, is pretty common and it wouldn't hurt to learn its many uses. You can read more about this here.

Photon torpedoes

You'll need two LEDs to represent the photon torpedoes. You could find rectangular LEDs or square off standard round LEDs to fit into the torpedo launchers.

Phasers

There are 12 phaser emitters on the refit, but we only have two pins dedicated to lighting them. If you decide to only light two, I suggest picking one above and one below. If you decide to route the phaser pin output through transistors, you could, of course, light several LEDs from each pin.

PWM lighting

Several of the light sources on the model are connected to the impulse and warp pins on the ATtiny, via either a transistor or a MOSFET. The deflector dish and the impulse deflection crystal are connected to both. The magnatomic amplification and the warp nacelles are connected to the warp pin. And the impulse engine is connected to the impulse pin. If you decide to use the 555 timer mentioned above, the window lighting would also be controlled by PWM.

Additional lighting

The additional lighting on the ship is of two types, fiber optic and individual LEDs. There is a docking port aft of the bridge, two on either side of the engineering hull, one on the port side of the saucer (it is rectangular instead of round). There are lights illuminating these ports and you'll probably have to rely on fiber optic to light them.The main shuttle bay entrance also has a number of lights that might also require fiber optic lighting or SMDs.

Shuttle bay landing lights

One feature that Enterprise-A modelers might want to include is the shuttle bay landing lights seen in the fifth Star Trek movie. This is basically just an LED chaser sequence that could be simulated with the 74HC595 that's already being used to light the RCS thrusters.

Your options are to re-purpose the 595 for this—lose the RCS thrusters to get the landing lights. You'll still need to free up an additional pin that's connected to a button if you want to trigger the effect, although you could just make this a random feature activated by the sketch.

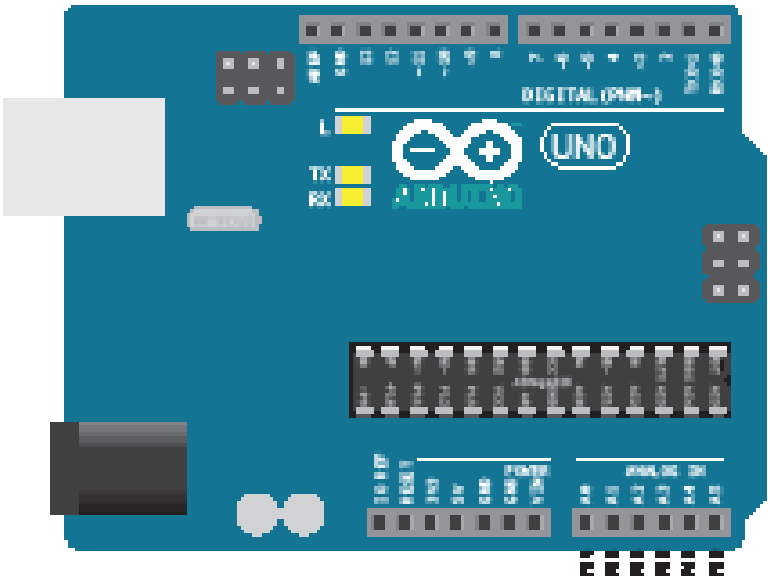
Or, you could daisy chain another 595 to the first, giving you and additional eight outputs. This doesn't require additional pins on the ATMega328 other than the aforementioned button pin. It's pretty simple and it's addressed in the More about chapter.

Another option is to power the landing lights with a 555 timer connected to 4017 decade counter. A decade counter can sequentially power its ten output pins based on a clock signal sent by the 555. There's more about the 4017 in the More about chapter.

Additional microprocessors

Many of the lighting conundrums presented here could, of course, be reconciled by adding more microprocessors or using larger microprocessors (with more input and output pins). Instead of an ATtiny and ATMega you could opt for two ATMegas. Or use an Arduino Nano, which although it's based on the ATMega328, uses the surface mount version of the chip that has additional pins.

So really it's up to you what features you want to include or what microprocessors you use. This guide is just a starting point.



We refer to the header pins on the Uno, but what you see is actually the female half of a Dupont connector, so technically they're holes.

More About ...

Before we begin, I thought I'd warn you about a common "gotcha" you might encounter when dealing with integrated circuits. This involves the numbering of the pins or legs on the ATMega328 and the ATtiny85. On an IC, legs or pins are numbered sequentially in an anti-clockwise fashion starting from the top left of a chip. There's usually an indent on one end of a chip to indicate the top, and often there's a circular depression to indicate pin one.

However, the physical numbering of pins rarely has anything to do with the pin assignments you'll specify when writing a sketch. In the Arduino Uno shown above, you'll see the numbered header pins at the top and bottom of the board. Digital input/output pins 13–0 are on top from left to right, and in the bottom right corner, you'll see digital I/O pins 14–19, which also serve as analog input pins A0–A5. These are the numbers you'll refer to in a sketch, not the physical layout numbers.

Looking at the ATMega328 chip on the Uno board, you'll see that physical pin one is in the upper right. The top of the chip is facing right. The second pin corresponds to digital I/O pin zero (which also serves as the receive pin) at the right of the top header row. Physical pin 14, in the bottom left corner of the chip, corresponds to digital I/O pin 8. Physical pin 28 is actually digital I/O pin 19 or analog input pin A5.

The Arduino IDE knows which pins do what based on the microprocessor you select. You'll see this process in the Programming the Microprocessor chapter.

Fortunately you can write a sketch that runs on the Arduino Uno, upload that sketch to an ATMega328, and it will run without a problem because the internal pin assignments are the same. That's not the case with the ATtiny85, and we'll look at that later in this chapter.

Arduino Uno

Let's take a closer look at the Arduino Uno. The gray box on the upper left is the USB port that connects the Uno to a computer. The Uno gets 5V from the USB cable. Below the USB

port at bottom left, you'll see the external power supply jack. You can power the Uno with a 7–12V external adapter (wall wart).

As mentioned before, you'll see the header pins across the top and bottom. The top row contains a GND pin and digital I/O pins 13–0. You can ignore the AREF pin. Just below the left side of the header pins you'll also see three LEDs. The top one is connected to pin 13, and that's the LED you'll see blink if you upload the beginner "Blink" sketch to the Uno. The other two LEDs correspond to the pins 0 (RX) and 1 (TX). You'll see those flash when you upload a sketch to the Uno.

The bottom header row includes reset, 3.3V, 5V, GND, Vin and the five analog pins. You can ignore the IOREF pin. Although there's a reset button on the Uno, it may be beneficial to have an external button wired to the pin, if you have enclosed the Uno. You won't need the 3.3V pin. The 5V pin is useful for supplying power to breadboards. And you can never have too many GND pins. Vin is only valid if you're using an external power supply, so if you're supplying the Uno 12V, you'll find 12V at this pin.

Arduino IDE

You will need to download the Arduino integrated development environment or IDE in order to program the Arduino. It's open source and free and should work with any Arduino, even those not made by the Arduino company. It can also program chips like the ATtiny85 or ATMega328. When you install the Arduino IDE, it will create an Arduino folder on your computer where you can add libraries such as the ones used here.

You'll need a USB cable to connect your computer with the Arduino, but that cable differs if you have the Arduino Uno (type A/B) or Arduino Nano (type A/mini-B).

ATMega328

ATMega328

As mentioned earlier, the microprocessor on the Arduino Uno is the ATMega328, and you might think why not just the Uno to power the starship. You certainly could, but a Uno costs $23. The chip is priced about $2. So, if you're lighting several starships, it makes economic sense to buy the chip. The chip also takes less room than the Uno, although admittedly once you've mounted the chip on a perfboard and added a power supply, that advantage disappears.

Connections

At a minimum on a breadboard or solderable perfboard, you will need to connect the reset pin (pin 1) to ground with a 10KΩ resistor; pins 7 and 8 to V+ and GND; and pins 21 and 22 to 5V. Because we're using an external clock crystal across pins 9 and 10, we'll also connect those pins to ground with 22pF ceramic capacitors. We could actually get by with even fewer connected pins and no external clock, but this arrangement works perfectly.

Numbered pins 2 and 3 correspond to pins 0 and 1 on the UNO. These are the serial read and write pins, so they're unused if you want to have serial communication with the ATMega. We don't care about this, so we're using these two pins as digital output to fire phasers.

Numbered pins 4–6 and 11–14 are digital output pins connected to the ULN2803A and control the activation of the spotlights. These pins correspond to pins 2–4 and 5–8 on the Uno.

Numbered pins 15 and 16 fire the torpedoes, and they're the same as pins 9 and 10 on the Uno. Notice on the picture of the chip there's a ~ next to the pin number, indicating these are PWM pins.

Numbered pins 17–19 connect to the Adafruit FX Sound Board. Pin 17 connects to the reset pin of the sound board, 18 to the RX pin of the sound board and 19 to the TX pin of the sound board. These numbered pins correspond to pins 11–13 on the Uno.

Numbered pins 23–28 are analog input pins, although we only use pin 28 for that purpose. They correspond to pins 14–19 on the Uno. Numbered pin 23 is used as a digital input connected to the button that triggers the weapons fire. Numbered pin 24 is used as a digital input connected to the Act pin of the sound board and detects whether a sound is playing.

Numbered pins 25–27 connect to the clock, latch and data pins of the 74HC595 and are all digital output. What's mysterious here is that pin 27 is not a PWM pin, and yet it's sending out serial data. I have no idea what voodoo makes this possible.

The last numbered pin is 28, and it's connected to the warp pin of the ATtiny85. Read more about that about in the chapter on PWM.

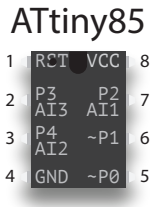
Of course, in our sketch, we don't refer to the physical pin numbering, but rather the numbers you see on the image above, within the body of the chip. However, one of the advantages of the ATMega328 is that a sketch written for the Uno runs without a problem on the ATMega because they use the same internal pin numbering.

Incidentally, there are a number of flavors of the ATMega328. Make sure you order the long, 28-pin version of the chip rather the square, surface mount version.

ATMega328			
1	Reset	Warp detect: AI5(P19)	28
2	P0(RX): phaser	data: AI4	27
3	P1(TX): phaser	latch: AI3	26
4	P2: spotlight	clock: AI2	25
5	P3~: spotlight	sound detect: AI1	24
6	P4: spotlight	weapons button: AI0	23
7	VCC	GND	22
8	GND	AREF	21

9	CRS: external clock	VCC	20
10	CRS: external clock	sound board TX: P13	19
11	P5~: spotlight	sound board RX: P12	18
12	P6~: spotlight	sound board reset: P11~	17
13	P7: spotlight	torpedo: P10~	16
14	P8: spotlight	torpedo: P9~	15

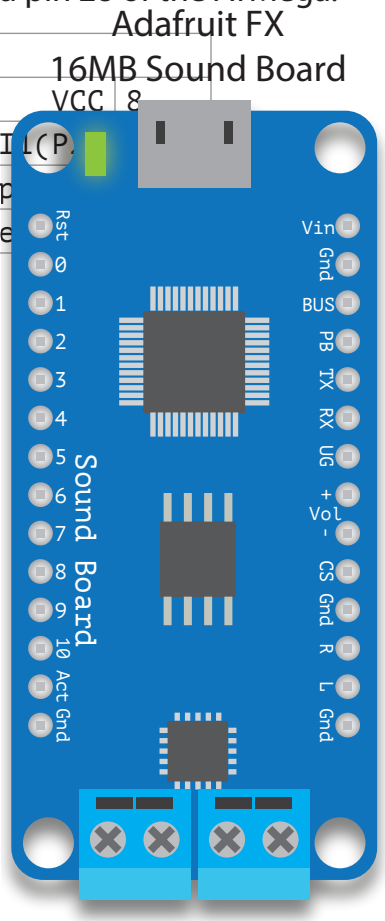
ATtiny85



This eight-pin chip has most of the capability of the ATMega, but with far fewer input and output pins. (It does has an 8MB RAM limit, compared to the 32MB limit of the ATMega.) Unfortunately, because of this, we can't use the same pin numbers in a sketch written for the Uno or ATMega328.

Numbered pin 1 is the Reset pin. Numbered pin 2 is actually either digital pin 3 or analog input pin 3 (A3) in a sketch, and it powers the navigation/formation lights. Numbered pin 3 is digital pin 4 or A2 in a sketch, and it is connected to the button that switches between impulse and warp lighting. Numbered pin 5 is PWM capable pin 0 and provides the impulse lighting signal. Numbered pin 6 is PWM capable pin 1 and is the signal for warp lighting. Finally, numbered pin 7 is either pin 2 or analog input A1. Numbered pin 8 connects to 5V. Numbered pin 6 also connects to numbered pin 28 of the ATMega.

ATtiny85		
1	Reset	
2	AI3(P3): nav lights	strobes: AI1(P
3	AI2(P4): warp button	warp
4	GND	impulse



Adafruit FX Sound Board

My use of the Adafruit FX Sound Board is one of the major differences between my lighting circuit and that of Trevor at Ostrich Longneck. He connects the ATMega to an SD card reader, which has an SD card loaded with sound effects. Although we're reading from the SD card, we're playing the sound using the audio output of the ATMega, which isn't very good. Sound effects like firing phasers and torpedoes is acceptable, but music sounds terrible.

Instead, I opted to go with the FX Sound Board, which is actually a stand-alone music player, capable of playing MP3 quality audio. It's available with either 2 or 16MB of onboard memory, and with/out an amplifier. I bought the 16MB ver-

sion with onboard amp.

The board can be used in trigger mode, where you connect buttons to 11 trigger pins, or in serial mode, where you connect a microprocessor to the RX and TX pins. We're using the latter option because we want to play sounds enabled by the sketch.

I'm not really sure which is the top or bottom of the board, so I'll just use identify the labels on the board. Vin (upper right) connects to 5V and GND connects to ground. You'll also want the UG pin to connect to GND, which tells the board to use UART or serial control. RX connects to numbered pin 18 on the ATMega328. TX connects to numbered pin 19 on the ATMega. The reset pin (upper left) connects to numbered pin 17 on the ATMega, and finally Act (bottom left) connects to numbered pin 24 on the ATMega.

The screw terminals at the bottom connect to external speakers. You can use the R and L pins to connect to speakers, but you'll need an amplifier. The volume pins don't work in UART mode, nor do the numbered pins on the left-hand side.

At the top of the board, you'll see a USB connector, which you can use to connect the board to your computer. The board will show up as a flash drive, to which you can transfer sound files. These sound files can be either in WAV format or OGG. The last may be unfamiliar, but it's actually an open-source format equivalent to MP3. WAV files play immediately when called by the sketch, but there can be a slight delay when play OGG files, because the file needs to be decompressed before playing. Fortunately, there's a lengthy tutorial on how to use the sound board.

One last little detail about the sound board—all sound files must be saved in the old 8.3 format familiar to MS-DOS users in the 1980s. So, the sound file that plays when the ship goes to warp is named ####WARP.WAV. The four hatch marks make up the four missing characters.

Audio clips

Most of the audio clips I used for my sound effects were downloaded from trekcore.com, which has a large number of clips including music from the movies and TV shows. Also, special effects like phasers, torpedoes and warp drive sounds.

I don't actually use the actual sounds from the first two movies for the phasers, torpedoes and going to warp. The phasers came from the original Star Trek series, the photon torpedoes from The Next Generation, going to warp from Voyager and dropping out of warp from The Next Generation.

The ambient bridge sounds came from Enterprise-B eight-hour clip on the ender4life channel on YouTube. The going to warp effect came from the thatSFXguy's channel on YouTube.

I also created a little dockyard performance using the digitized voices on my Mac and the voices found at the IBM Watson Text to Speech demo. The latter is free and you sim-

ply paste in the text you want the synthesized voices to say. I also did the same with the high-quality synthesized voices that come with the Mac OS (they have to be downloaded separately). In order to record the spoken text, I downloaded the SoundFlower plug-in that makes it possible to record internal audio. I arranged the audio clips in Adobe Audition, but you could download the free program Audacity.

PWM

Pulse width modulation is a trick used to turn digital signals (either on or off, 0 or 5V) into something like analog values. We use PWM to make LEDs gradually fade up and down in brightness for both the deflector glow and warp nacelle glow effect.

The microprocessor achieves the effect of fading up and down by rapidly turning on and off the 5V signal on certain pins. For instance, if you want an LED to be 50% bright, you'd turn on and off the 5V signal for equal periods of time. Obviously, however, if you turned on the LED for one second and off for one second, you would perceive that as blinking on and off. If you turn the LED on for 1 millisecond and off for 1 millisecond, you just begin to perceive this as 50% brightness instead of rapid blinking. So, to fade up, we increase the on time and to fade down we decrease it. Or rather the microprocessor does this. We just need to write code that progressively changes the output value of the pin.

PWM offers 256 levels of brightness, using numbers from 0 to 255. Here's an example:

```
analogWrite(9, 127);
```

We're telling pin 9 to send out a signal that will equate to 50% brightness.

On the ATtiny85, two pins—0 and 1 (numbered pins 5 and 6)—are capable of PWM, and on the ATMega328, six pins have this—3, 5, 6, 9, 10 and 11 (numbered pins 5, 11, 12, 15, 16 and 17). You can still use these pins to send a purely digital signal:

```
digitalWrite(9, HIGH);
digitalWrite(11, LOW);
```

Analog input

Sort of the flip side of PWM is analog input. Certain pins can read an analog signal—voltage—and as the voltage varies, assign it a value between 0 and 1023. On the ATtiny85, three pins—2, 3 and 4 (numbered pins 2, 3 and 7) are capable of reading an analog signal, and on the ATMega328, six pins—A0, A1, A2, A3, A4 and A5 (numbered pins 23–38). The code to read an analog input looks like this:

```
value = analogRead(A0);
```

Notice I typed `A0` for the pin number. If I wanted to read a digital signal on the numbered 23rd pin on the ATMega328, I would type:

```
value = digitalRead(14);
```

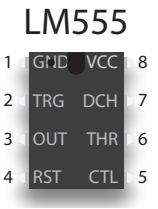
And value would be set to either `HIGH` or `LOW`. But numbered pin 23 is referred to as `A0` when I want to analog read that pin.

There's only one instance of analogRead in our starship lighting circuit, and it's pretty unusual. We use the ATtiny85 to change from impulse (amber deflector lighting) to warp (blue deflector and warp nacelle lighting), controlled by a button connected to pin 4, numbered pin 3 on the ATtiny. Pins 0 and 1, used in PWM mode, control the amber and blue LEDs that signify impulse and warp.

The ATMega328 controls the sound effects of our lighting circuit, and it needs to know when someone has pressed the impulse/warp button connected to the ATtiny in order to play the appropriate sound effect. We do that with pin A5 (numbered pin 28) on the Mega, using it to detect when pin 1 (numbered pin 6) above or below certain values.

Unfortunately, this is a little tricky. We're trying to use an analog read pin on the Mega to detect the value of a PWM pin on the Tiny. Remember, if the PWM pin is sending out a 50% signal, half the time it's sending out 5V and half the time 0V. So, the analog read can get very confused what is the actual value of the Tiny pin. We get around that by tying pin 28 on the Mega to ground with an electrolytic capacitor. I used a 100µf capacitor, which has the effect of blurring the signal enough (by constantly gradually charging and discharging) that the value can be read.

555 timer



The venerable 555 timer (introduced in 1971) is used in millions of places and before small microprocessors like the Arduino supplied the heartbeat of many a starship model. It can be used in three different modes—monostable, bistable and astable—but for our purposes we want to use it as an astable oscillator.

Going through the pins from top left, we have:

1. GND, which connects to the ground of the power supply
2. TRG, the trigger pin. In monostable or bistable mode, the trigger can be connected to a physical switch, but more likely to the output pin of another chip. In astable mode, pin 2 is connected to pin 6, because we're using the value of the threshold pin to "pull" the trigger.
3. OUT, or output is the HIGH or LOW signal produced by the chip
4. RST, or reset, should be connected to V+ to prevent reset, so you'll almost always see pin 4 connected to pin 8
5. CTRL, or control, is rarely used. It usually goes to ground through a smoothing capacitor to prevent noise in the circuit.
6. THR, or threshold, when it reaches a certain voltage, pulls the trigger in astable mode. It's attached to a capacitor that gradually builds up the voltage that exceeds the threshold value. The speed at which the capacitor charges is controlled by a resistor connected to pin 7. If we're using the 555 as a PWM LED dimmer controller, we're using a potentiometer instead of a resistor.
7. DCH, or discharge, is connected in astable mode to pin 8 with a resistor, and to pin

6 with another resistor. It's the value of the resistors and the capacitor connected to pins 8, 7 and 6 that control the frequency of the clock signal produced by the 555 and the duty cycle of the signal—what percentage of the output is HIGH or LOW.

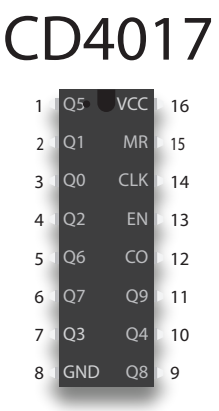
8. VCC connects to the 5V power supply. Some users place a smoothing capacitor between GND and VCC, but it's not necessary here.

I don't have the knowledge to explain how the 555 works, any more than I can explain how the ATtiny or ATMega works. But I have read or watched a lot of explanations of how the 555 works, and although I don't completely understand it, it has helped. Even a vague, fuzzy sort of knowledge makes it easier to understand what the pins do and how to wire them.

Beyond using the 555 as a PWM dimmer, it can be used to produce many of the basic starship lighting effects. Two 555s—or more efficiently a 556, which is two 555s on a single 14-pin chip—can power collision strobes and navigation lights. Just like the ATtiny85, however, the outputs should control transistors that switch the various LEDs to a separate power rail. The total output of a 555 is only 200mA, which theoretically could power up to 10 LEDs, but the chip would probably run hot.

Another 555 coupled with a 4017 decade counter could also produce the spinning Bussard collector lights on the original enterprise. Or a 556 could power both the lights and drive stepper motors to recreate the spinning fan blade effects.

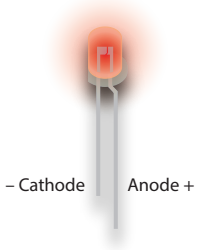
4017 decade counter



If you'd like to add the shuttle bay landing lights effects seen in the fifth Star Trek movie, you have a few options as mentioned in the Lights and Sound chapter. The easiest solution is to pair a 555 timer with the 4017 decade counter. You can see the complete circuit in the Examining the Board(s) chapter, but there are some complexities of the 4017 we'll examine here.

Every time the 4017 reads a HIGH signal on the clock pin, it sequentially powers the ten output pins. The output pins are arranged haphazardly, as you can see in the pinout. If you need less than ten outputs, you'll want to connect the unused pin to reset pin 15 to avoid a pause. So if you only need to light eight pins, tie pin 9 (Q8) to pin 15

LED basics



LEDs bring our starship model to life and you will use a couple hundred of them in your refit model. Most of the LEDs you use will be in the form of LED tape—a string of LEDs that are typically powered by a 12V power supply. But you'll also use individual LEDs for the collision strobes and navigation lights and the floodlights.

LEDs, of course, are light-emitting diodes and diodes have a polarity. Positive voltage must be connected to the anode (think "A-plus") pin and ground to the cathode. The anode leg is usually longer and sometimes has a 45° kink just under the bulb. The cathode is shorter and sometimes the rim of the bulb has a flat spot on that side.

An output pin from the Arduino is supplying positive voltage, but sometimes that pin is connected to something like a transistor, and the "signal" coming from that transistor to the LED is the ground. This is the case for our collision strobes, navigation lights and the MOSFET that powers the warp nacelles. So, in this case, the signal from the transistor is attached to the ground leg.

LEDs almost always need to be connected to a limiting resistor or else they'll burn out. The value of the resistor depends on the color of the LEDs and the voltage supplied to the LED. You can use Ohm's Law to determine what resistor value to use, but frankly I just use online calculators to figure this out. An online calculator will ask for the voltage (always 5V in our circuit), the voltage drop of the LED (depends on the color and brightness), the current draw (usually 20 milliamps) and the number of LEDs, which is always one. Unfortunately, I've acquired LEDs here and there, so I'm not certain of the exact voltage drop or current draw of an LED, so I often use a higher value resistor than the one suggested by online calculators.

It always one because although we're wiring LEDs in parallel, we're still attaching a resistor to each LED. Don't be tempted like I was to wire LEDs in parallel but use a single, low-value resistor for all the LEDs. Should one of the LEDs fail, then the remaining LEDs will run a little hotter, and then one of those LEDs will fail, and so on until they all fail.

By the way, most people solder connectors to the ground leg of an LED, but in fact you can solder it to either leg.

LED tape, incidentally, has built-in resistors that are perfect with a 12V supply, but still work, if not quite as bright, with 9V. LED tape, interestingly, are three LEDs connected in serial, but each three-LED group is wired in parallel. If one LED fails, all three LEDs won't light, but the rest of the LEDs still work. This is why you can only cut LED tape every three LEDs.

To make my life easier, I often buy white LEDs and color them as needed. This way I can use the same resistor value for all. LED bulbs are so slick, however, they can be hard to color. It's best to sand the bulbs with a fine grit sanding stick or spray a matte lacquer. Then I dip them in clear, colored acrylic or apply a permanent marker of the appropriate color, followed by a clear acrylic topcoat.

1N4001 Diode

Diodes are very similar to LEDs, in that they're a solid-state device that permits electrons to only flow in one direction. We use a diode in our on-board power supply to prevent connecting a power supply (the AC/DC adapter) with reverse polarity. Most adapters will have a power sup-

ply with the tip or inner barrel supplying positive voltage, but there are some exceptions. Incidentally, LEDs can't be used for this purpose because they usually will conduct if the reverse voltage is high enough. The 4001 diode we're using can withstand a reverse voltage of 50V and withstand 1A current draw.

1st stripe	2nd stripe	3rd stripe (if 4 stripes total)	3rd stripe (if 5 stripes total)	4th stripe (if 5 stripes total)	Tolerance (last) stripe
Black = 0	Black = 0	Black = 1	Black = 0	Black = 1	
Brown = 1	Brown = 1	Brown = 10	Brown = 1	Brown = 10	Brown = 1%
Red = 2	Red = 2	Red = 100	Red = 2	Red = 100	Red = 2%
Orange = 3	Orange = 3	Orange = 1K	Orange = 3	Orange = 1K	Gold = 5%
Yellow = 4	Yellow = 4	Yellow = 10K	Yellow = 4	Yellow = 10K	Silver = 10%
Green = 5	Green = 5	Green = 100K	Green = 5	Green = 100K	
Blue = 6	Blue = 6	Blue = 1M	Blue = 6	Blue = 1M	
Violet = 7	Violet = 7	Violet (not used)	Violet = 7	Violet (not used)	
Gray = 8	Gray = 8	Gold = 0.1	Gray = 8	Gold = 0.1	
White = 9	White = 9	White = 0.01	White = 9	White = 0.01	

Resistor basics

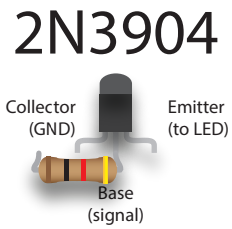
Resistors, which reduce current flow, are common components of any circuit. In our starship lighting circuit, they're connected to every LED, but you'll also find them connected to every button, every transistor, to the reset leg of the ATMega328 and to the MOSFET for the warp nacelle glow. You will only need ¼ watt resistors for this circuit and 5% tolerance is acceptable, especially if you always guess a little high for your resistor values.

Resistors are rated in ohms (Ω) and reading the value of a resistor is easy if you can remember the rules, but I often use online calculators or the multimeter to confirm. Colored stripes on the resistor represent its value, and most resistors have three stripes indicating the value with a fourth the tolerance of the resistor. Some resistors have four stripes for the value and the fifth is the tolerance. On a four-stripe resistor, the first stripe represents the first value, the second stripe the second value and the third stripe is a multiplier. On a five-stripe resistor, the third stripe is a value and the fourth is the multiplier. There is a small gap between the last value and the tolerance stripe.

So, for a four-stripe resistor with brown, black, red and gold stripes, you can read that as a 1 KΩ resistor—1 and 0 times 100—with 5% tolerance, meaning the resistor could be as low as 950Ω or as high as 1.05KΩ.

It can be quite difficult to read a resistor—red, orange and yellow look similar—so you may need to use the multimeter to check these values. Fortunately, there are online calculators that will give you the colors for a given value or that will let you pick colors for each stripe and return the value for that combination.

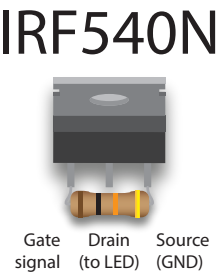
2N3904 transistor



The ATtiny85 and ATMega328 microprocessors can only supply a small current to the output pins—not enough to light more than two LEDs. So, we need a way to connect a large number of LEDs to a larger power supply, using a signal from a pin on the microprocessor to activate a switch that makes that connection. Enter the 2N3904 transistor, a ridiculously cheap (50¢) component that acts as that switch. A signal sent from one of the microprocessor pins to the base leg of the transistor will connect other components like LEDs to a higher voltage. The signal voltage doesn't need to be more than 0.6V, but we're sending 5V, so there's a 1KΩ resistor between the microprocessor and the base leg of the transistor.

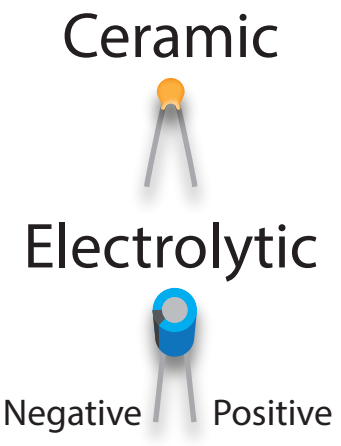
The LEDs we want to light are connected to the emitter leg of the transistor, which is actually a path to ground via the collector leg of the transmitter. So, connect the emitter leg of the transistor to the cathode or ground leg of the LED, and the anode or positive leg of the LED to the higher voltage. In this case, however, we use a 5V signal from the microprocessor to the transistor, which switches the LEDs to the same 5V voltage that went into the microprocessor. But importantly, we're not sending current through the microprocessor to those LEDs, we're getting it directly from the AC/DC power supply adapter (wallwart).

IRF540N



A MOSFET is another solid-state device, and for our purposes, we're using it like a super-duper switching transistor. The pin layout of the IRF540N MOSFET we're using is different from the 2N3904 transistor, but the idea is the same. We're sending a 5V signal to the gate leg of the 540N, and the LEDs we're driving (in our case, the LED strips in the two nacelles) are connected to the drain leg. The ground pin of the strips is connected to the 540N and the positive pin of the LED strip is connected to our 12V supply. There's a pull-down 10KΩ resistor bridging the gate and source legs of the 540N which prevents accidental switching.

Capacitors



Capacitors store and release energy, sort of like a battery, although they can store and discharge quite quickly. Anyone who has inadvertently shorted a big capacitor in some sort of appliance like a television knows you can end up with a melted screwdriver and/or loss of consciousness. Fortunately, the value of the capacitors we're using are unlikely to cause harm.

We use both ceramic and electrolytic capacitors in our board. Ceramic capacitors resemble little blobs of clay with two metal

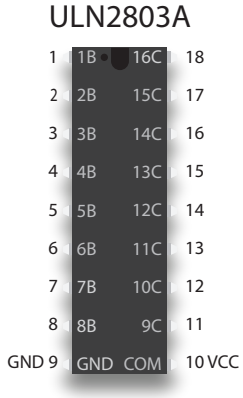
legs sticking out at a jaunty angle. The smaller value capacitors we use are ceramic, the larger ones are electrolytic. Unlike the ceramic, electrolytic capacitors are polarized—they have a positive and negative leg. Make sure the negative leg goes to ground. The negative leg is usually indicated by a dark stripe with white bars.

The value of a capacitor is rated in farads, however, a one farad capacitor would be a honking, big capacitor. Most of the electrolytic farads you'll use for starship lighting are measured in microfarads. There's a 10µF (microfarad/1/1000th of a farad) electrolytic capacitor we use to bridge the reset and ground pin on the Arduino Uno to keep it from accidentally resetting when using it as an in systems programmer. We use ceramic, 22 pF (picofarad/1 trillionth of a farad) capacitors between the legs of the ATMega32 clock crystal and ground. Electrolytic capacitors also have a voltage value, and for our board 16V would be fine. Larger values are also physically larger.

Electrolytic capacitors are generally clearly labeled with its capacitance and voltage, but ceramic capacitors requiring deciphering a code that I think is even more cryptic than resistors. Most ceramic capacitors generally have a two- or three-digit code. The first two digits are the capacitance in picofarads. The third digit is a multiplier and the value represents how many zeroes after the one. So a capacitor labeled with 104 is both 100,000 pF, 10 nF and 0.1uF. The 22 pF capacitors I used between the clock crystal and GND on the ATMega328 are just labeled 22, but they could also be labeled 220.

Incidentally, the values of resistors and capacitors are quite odd because they're spaced logarithmically. That's why online resistor calculators usually recommend the next highest common value.

ULN2803A

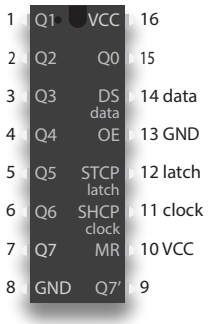


In an effort to further reduce the load on the microprocessor, we're going to offload the lighting of the spotlights to the ULN2803A chip, a Darlington array. We can connect seven of the output pins of the ATMega328 to seven of the pins on the 2803 (the 2803 actually has eight input/output pins, but we only need seven). The 2803A can send up to 500 milliamps out through each pin, but we need only a little of that.

The input pins are on the left side of the chip, the output pins on the right. (This chip is an anomaly because the layout of the pins makes sense.) Pin 9 (lower left) is connected to ground and pin 10 (lower right) is connected to our 5V power rail. Unfortunately, using the 2803A still ties up seven pins on the ATMega. Luckily, the next solution only uses up three pins while lighting eight sets of LEDs.

74HC595

74HC595



The 74HC595 chip is a shift register, it can convert a serial signal—of values from 0 to 255—coming in and turn it into up to eight separate 5V HIGH outputs. We only need three pins on the ATMega connecting to what are commonly called the data, latch and clock pins on the 595. The data pin receives the data that tells the 595 which of the output pins to light up.

That data is a little complicated because of what we're trying to do. We want to simulate the effect of the reaction control system that are scattered about the ship. To fire the thrusters, we're going to put the values that will light the thrusters into an array (more on that when we look at the actual sketch) and randomly retrieve one of these value at random times.

But in one of the most bass-ackward processes you can possibly imagine, we're going to send a base 10 number and turn it into its equivalent 8-digit binary value. So, for instance, if we send the base-10 number 160 to the 74HC595, it will be turned into the equivalent binary number 10100000. Those eight digits equates to the eight output legs of the 595, and a one means send out 5V, and a zero means OV. So, two of the eight LEDs will light.

If we send the base-10 number 40 to the 595, then the equivalent binary number of 101000 will light another two RCS LEDs. You might notice that number is only six digits long, so the 595 will just add another two zeros at the beginning to give 00101000. In fact, all the values we'll put in the array will only light two LEDs at a time, which is intentional. The 595 can only drive no more three LEDs at full intensity, although it can drive more lights at lesser intensity or for short durations. To play it safe, however, I'm limiting it to two "random" RCS at a time.

Notice that in addition to the number 8 ground pin and the number 16 incoming voltage pins, we will want to ground pin 13 and connect power to pin 10. Pin 13 is the output enable pin. If it went high (connected to power), the 595 would not light any LEDs. Pin 10 is the reset pin and if it goes low (connected to ground), the chip resets.

Setting the pins to go high involves three lines of code:

```
digitalWrite(LTCH_PIN, LOW);
shiftOut (DATA_PIN, CLK_PIN, MSBFIRST, 0);
digitalWrite(LTCH_PIN, HIGH);
```

The first line sets the latch pin low, which makes the chip ready to accept data. The next line sends data—in this case zero—to the data pin of the chip. The third line sets the latch pin high, which actually sets the desired pins high.

When to use ULN2803A and 74HC595

There is a trade-off between the ULN2803A and 74HC595. The first chip can light up to eight LEDs at a time but requires using eight pins of the microprocessor. The second chip

only uses three pins of the microprocessor but shouldn't be used to run more than two or three LEDs before exceeding the maximum current output of the chip. The 2803 has the additional feature that it can power multiple LEDs per pin. The 595, on the other hand, has the feature that you can daisy chain additional 595s without requiring the use of more pins on the microprocessor, but each 595 still has the same current limitations.

LM7805

Our onboard power supply uses the LM7805 voltage regulator to step down the 12V input from the AC/DC adapter to the common 5V we supply all the components on the board. Just the like the IRF540N MOSFET, the 7805 has three legs, one attached to the 12V supply, one attached to ground and one that supplies 5V to the rest of the board. It's not a very efficient regulator—the excess voltage is disposed of as heat, which is why it's best to attach a heat sink to the back of the 7805—but it is simple. You could get by with just the 7805, but it's best to attach some smoothing capacitors that bridge the legs of the 7805. We use a 100µF electrolytic capacitor across the input and middle ground pin, and a 10µF electrolytic capacitor across the output and ground pin. These capacitors ensure a stable 5V output.

JK16-050T resettable switch

This component, which resembles a ceramic capacitor, prevents the power supply from drawing more than 500mA current. It's sort of a fuse, except that it can be triggered repeatedly (although that's not a good thing). The switch isn't polarized and is placed just after the 1N4001 diode between the 12V supply and the input pin of the 7805.

You'll know it's been triggered because the board has no power, and the switch is hot to the touch. Unfortunately, it's not perfect protection because it's not triggered instantaneously, and in that period some components might be damaged.

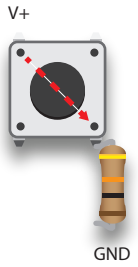
16Mhz clock crystal

This is not really a crucial component, but it does ensure the ATMega328 operates exactly at 16Mhz. If we had a lot of crucial timing in our sketch, it would be essential to attach a clock crystal to the ATMega, rather than depend on its internal crystal. If you look at an Arduino Uno, you'll see such a crystal. Since the crystal doesn't take up much space and is pretty cheap, it's best to add the crystal across numbered pins 9 and 10. We also should add the 22pF ceramic capacitors mentioned earlier.

If you elect to use the ATMega328 without the crystal, you'll need to install a different

bootloader. Incidentally, we do depend on the internal clock of the ATtiny85, primarily because there is no option to use an external clock.

Buttons



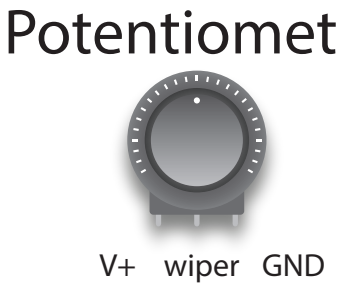
You'd think buttons were simple, but they actually are kind of complicated. We're using momentary buttons to trigger the change from impulse to warp, and to fire weapons. These buttons only fire when the button is depressed, but we don't want to trigger functions in the sketch repetitively while the button is depressed. We don't want to rapidly go from impulse to warp and back again several times, for instance, so we need to set software flags to prevent this.

The most common momentary buttons have four legs and are meant to straddle the horizontal gap of a breadboard, although it doesn't have to. If you look closely at the picture above, you'll see that a positive voltage connected to the upper left pin will be transmitted to the pin in the lower left when the button is pressed. You should also notice a 10KΩ pull-down resistor that goes from that pin to ground. The resistor filters out electrical noise that might be interpreted by a microprocessor as a pressed button.

You should also know that even though there isn't a connection from upper left to lower right until the switch is closed, there is always a connection from the upper left pin to the lower left, and the same holds true of the pins on the right side.

I actually bought two pin, "breadboard friendly" momentary buttons. The pins on most momentary buttons are actually a bit too wide to easily fit in the holes of a breadboard, but the buttons I bought fit perfectly.

Potentiometer



A potentiometer is a variable resistor that we use to control the 555 PWM LED dimmer. We could also have connected potentiometers to our microprocessors to fine tune the timing of the collision strobes or navigation lights, or the speed at which the deflect dish glows—except we don't have any free pins for that purpose.

Potentiometers, or pots, have three pins and usually the middle pin is the wiper or the variable resistance value. The outside pins can be thought of as the input voltage and ground, but that's not very clear in our 555 PWM board. It doesn't really matter which outside pin you use for what, but it does change how the pot works, making the LEDs brighter as you turn clockwise, or dimmer as you turn clockwise.

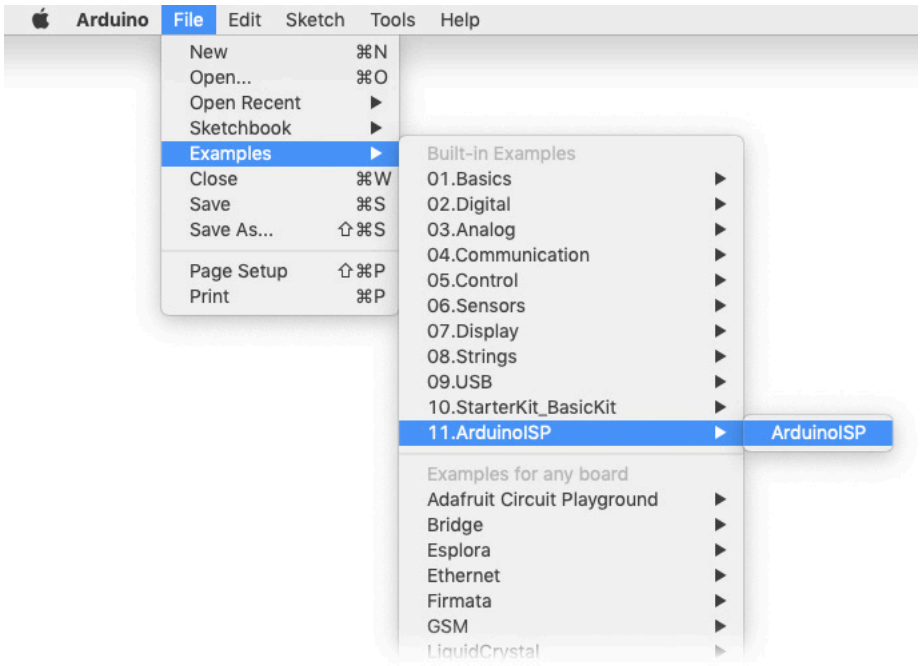
The pot shown on the 555 PWM board is a rotary potentiometer, but I plan to use a slide potentiometer, reminiscent of the transporter controls on the original Star Trek show. I also plan to a use stereo slide potentiometer to adjust the volume of the speakers. This is essen-

Programming the ATtiny85 and ATMega328

There's no way to sugar coat this: Using the Arduino Uno to program other microprocessors is pretty complicated and it's easy to make a mistake. There are actually several ways to program another microprocessor, but the method I'll describe below is the least complicated. NOTE: Although the screenshots below are of the Mac OS version, it's layout and operation is identical to the PC version.

Step 1: Open In System Programmer sketch

Drop down the **File** menu and choose Examples. In the sub menu that pops up, go to **11.ArduinoISP** and open the sketch called **ArduinoISP**.



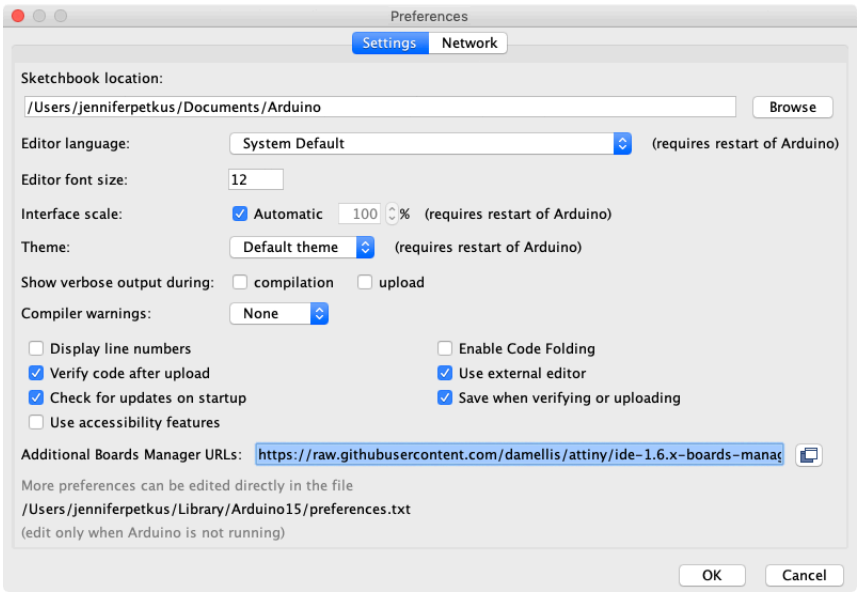
If you upload this sketch to the Arduino Uno, it will turn the Uno into an In System Programmer—in other words, it turns the Uno into a device that can program microprocessors like the ATtiny85 and ATMega328. By the way, just like any sketch you upload to the Uno, this sketch will remain in the Uno even after you unplug it from power. There's no need to re-upload this sketch each time you use the Uno as a programmer unless you've uploaded a different sketch in

the interim.

Step 2: Target the microprocessor

If you want to program the ATtiny85, you'll first need to give the Arduino IDE some information. Go to **Preferences** in the **File** menu and look for the line **Additional Boards Manager URLs:**

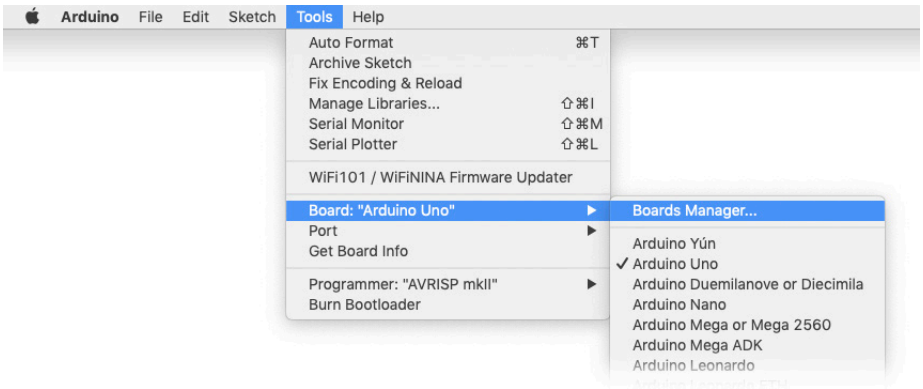
Paste in the URL you see below the picture to the right (you can just copy that text) into the blank field and click OK.



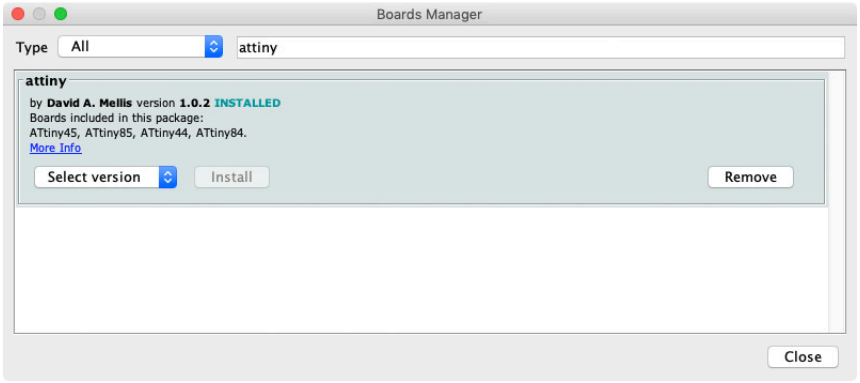
https://raw.githubusercontent.com/damellis/attiny/ide-1.6.x-boards-manager/package_damellis_attiny_index.json

You've identified the location of the file that describes the ATtiny family of processors, but now you need to tell the Arduino IDE to install it.

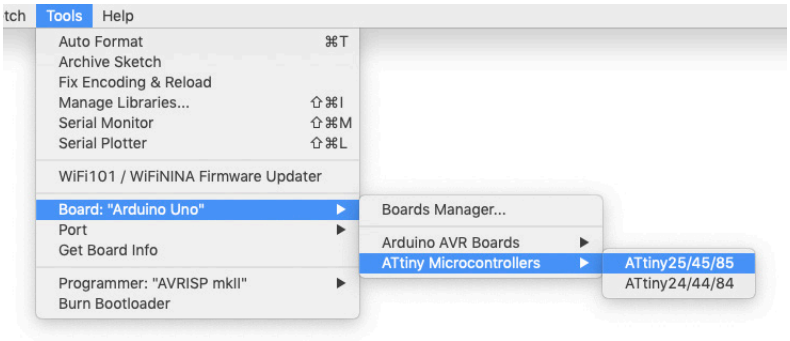
Next, choose **Boards Manager…** in the **Tools** menu. You'll find it in a drop-down menu next to the currently targeted processor, which will probably be **Board: "Arduino Uno"**



It can be difficult to find what you're looking for, so type "attiny" in the search field and you'll see the manager for the ATtiny family. Select the newest version of the manager (the picture shows v 1.0.2) and click **Install**.

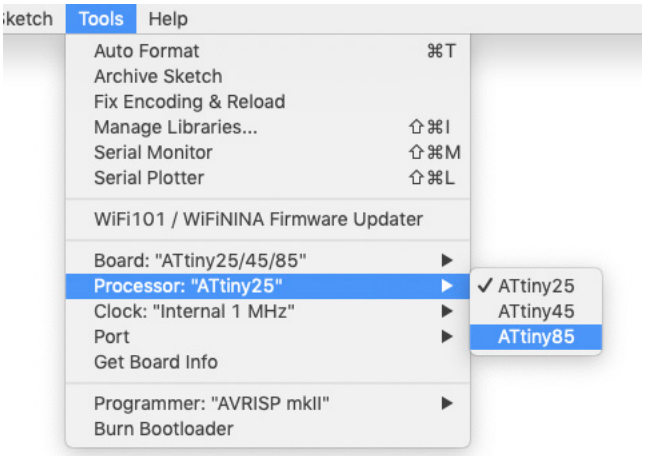


OK, step 2 isn't done yet. From the **Tools** menu again, drop down the menu that probably says **Board: "Arduino Uno"** and then drop down the sub menu that says **ATtiny Microcontrollers**. From that submenu, choose **ATtiny25/45/85**.

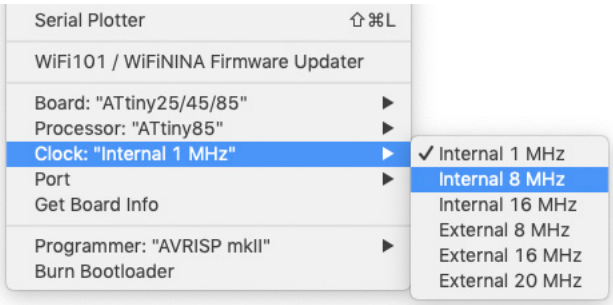


Nope, still not done. A new menu item has appeared in the **Tools** menu, just under the menu item that now says **Board: "ATtiny25/45/85"**

The new menu item is called **Processor: "ATtiny25"** Click it to drop down a sub menu where you'll choose **ATtiny85**.

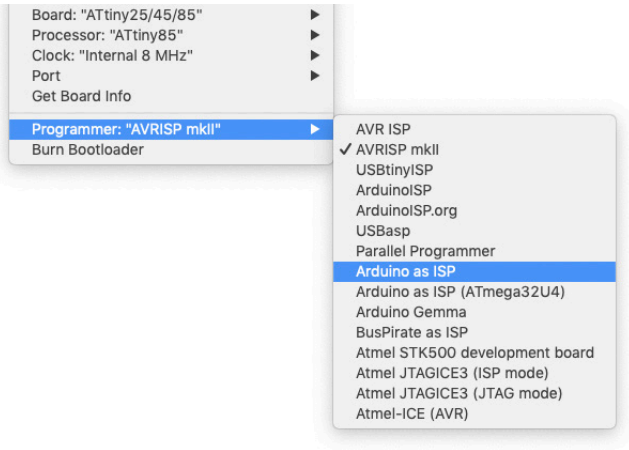


Nope, still not done. You must now make your choice for the clock speed of the ATtiny85. I think you could leave this at the default 1MHz, but I think you're better off picking a clock speed closer to the clock speed of the Uno you've used for development. However, you should avoid the **Internal 16MHz** option (or any of the external options) unless you really know what you're doing. Which leaves you to choose **Internal 8 MHz**.



And now you're done … with this step. The good news is that *if you're programming the ATMega328, you don't need or want to change the board from the Arduino Uno.* The ATMega328, is after all, the same processor as on the Uno, although there might be some extra steps required if you skip the external clock for the ATMega.

Step 3: Tell the Uno to be an In System Programmer



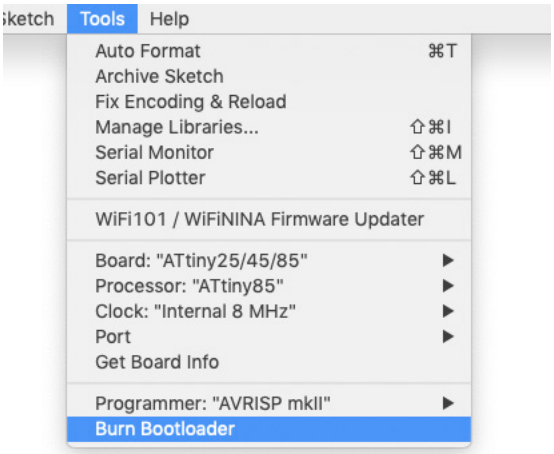
But wait, I hear you wail. We already did that in step 1! Yes, you uploaded the In System Programmer sketch to the Arduino, but now you have to tell it you really mean it. Which you will do by returning to the **Tools** menu and dropping down the menu item that probably says **Programmer: "AVRISP mkII"** From the sub menu, choose **Arduino as ISP**.

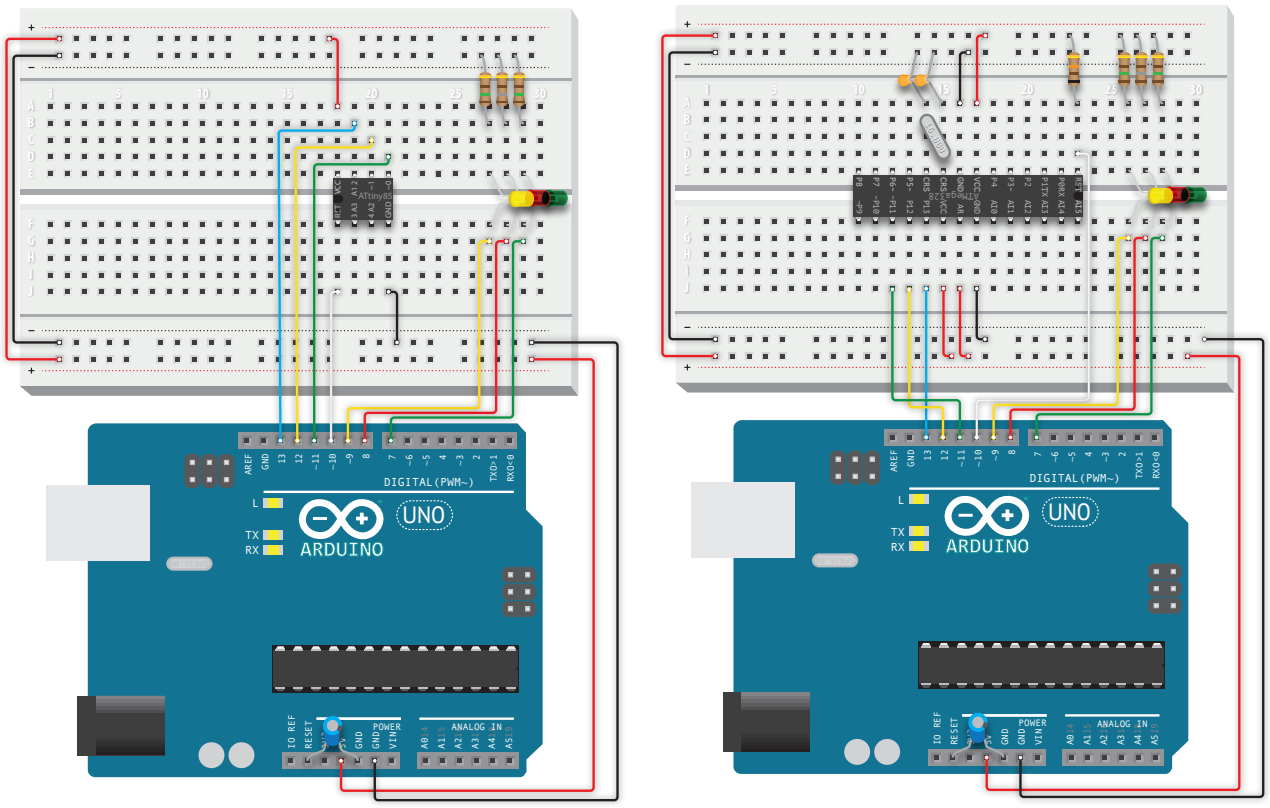
Don't get seduced by any of the other options. Pick **Arduino as ISP**. The good news is that you won't need to change this selection, even if you go back to normally programming your Uno.

Step 4: Burn the bootloader

This step might not actually be necessary. Often when you buy a microprocessor like the ATtiny or ATMega, a bootloader has already been burned onto the chip. Without the bootloader, it's just a dumb brick and doesn't know what to do with any sketches you might want to upload it.

You'll only need to burn the bootloader once, although it's possible you might do something that corrupts the bootloader, and you may have to upload it again. You'll find **Burn Bootloader** at the bottom of the **Tools** menu.





Step 5: Connect the Uno to the microprocessor

Yes, I put the cart before the horse. You can't burn the bootloader or upload a sketch to the microprocessor until you've connected it to the Uno. I have a half-size breadboard that I use for programming a chip, so I don't have to tear up whatever is on the main board whenever I need to upload a sketch. Note that in the example above, the ATtiny is facing left and the ATMega is facing right. It was just easier to draw this way.

The process of uploading to either the ATtiny or ATMega is similar. I've added some status lights to the breadboard that indicates the progress of an upload. These LEDs are connected to pins 9, 8 and 7 on the Uno. The yellow LED connected to pin 9 is the heartbeat, which pulses whenever the Uno makes a valid connection to a microprocessor. The red LED indicates a problem, and the green LED shows an upload in process.

There are 150KΩ capacitors connecting the yellow and green status LEDs to ground and a 180KΩ resistor connected to the red LED, but you could just use the 180KΩ value for all three.

Pins 13, 12, 11 and 10 of the Uno are connected to either numbered pins 7, 6, 5 and 1 on the ATtiny85, or numbered pins 19, 18, 17 and 1 on the ATMega328. Notice pin number 1 (reset) on the ATMega is connected to ground with a 10KΩ resistor—that's probably not necessary for the ATtiny.

The ATMega's connections are more complicated, and you'll see the 16MHz clock crystal across numbered pins 9 and 10 and the two, 22pF ceramic capacitors connecting to

ground. Also notice the 10µF electrolytic capacitor bridging reset and GND on the Uno. This prevents accidental resets, which I'm told this has been fixed on newer Unos, but apparently not on mine.

Power to the microprocessor comes from the 5V and GND pins on the Uno.

Step 6: Upload final sketch to microprocessor

After you've burned the bootloader to the microprocessor, you're now ready to upload your sketch to the microprocessor, but there's one final GOTCHA! If you were to simply click on the arrow to upload your sketch, you'd actually be uploading it to the Uno and replacing the ArduinoISP sketch you originally uploaded all the way back in step 1. Instead, you need to hold down the SHIFT key before clicking the arrow. You'll see the label Upload change to Upload Using Programmer. (In the **Sketch** menu of the IDE, you'll also see options for **Upload** and **Uploading Using Programmer**.)

And that's all there is to it! Every time you make a change to your sketch, of course, you'll need to lather, rinse and repeat. Fortunately, the IDE will remain in whatever state you last used it, unless you change processors or forget to hold down the SHIFT key when uploading a sketch to a processor.

If you forget one of these steps, you'll see error messages saying you've done something wrong. Common mistakes I make include forgetting to upload the ArduinoISP sketch to the Uno before trying to upload to the microprocessor, forgetting to hold down SHIFT when I want to upload to the microprocessor, forgetting to add the capacitor across reset and GND, forgetting that the ATtiny85 is still selected when I'm trying to upload to the ATMega328 (and vice versa) and accidentally dislodging some jumper wire on the breadboard.

Fortunately, practice does make this process easier and I rarely make these mistakes … unless it's been a couple of months since the last time I needed to do this.

Examining the sketches that drive the starship

strobes_navs_deflector_ATtiny.ino

There are two sketches that power our starship and two microprocessors that run those sketches. They largely operate independently of each other, although the sketch on the ATtiny85 is timed to coincide with the startup sequence of the sketch on the ATMega328, and the ATMega sketch reads the status of a pin on the ATtiny to know whether the ship is at impulse or warp.

LedFlasher and arduino-timer

Before we begin examining the ATtiny sketch, let's look at two libraries downloaded to make blinking lights easier to create. The Arduino IDE or integrated development environment comes with many libraries pre-installed and you may be unaware of where these libraries are located. You can access these libraries by writing an `#include` directive at the top of your sketch. But you can add your own libraries by downloading them into the Arduino folder created when the IDE was installed. In the Arduino folder, you'll need to create another folder called libraries and within that, place the folders and files extracted from the download.

The first library, LedFlasher, created by Nick Gammon, is simple but useful. It just blinks lights and is used to flash the collision strobes and the navigation/formation lights. After the `#include` directive, two instances of the class are created and named. The arguments in the parentheses declare which pins to use as outputs; how long, in milliseconds, the amount of time an LED is off; and how long it's on. *Note: You don't have to declare these pins as outputs with a* `pinMode()` *declaration.*

```
#include <LedFlasher.h>
#include <arduino-timer.h>
LedFlasher strobeLights (2, 900, 100);
LedFlasher navLights (3, 500, 1000);
```

In the `setup` block, I make a call to the `begin()` method of the class for each instance, and finally in the main loop block, I have a call to the `update()` method for each instance. And that's it! Easy peasy.

The second library, arduino-timer, created by Michael Contreras, is a little more complicated. Both libraries make it possible to do repetitive actions without resorting to delay statements to manage the time between these actions. The classic blink.ino sample you should be familiar with uses a delay between the on and off state and another delay after the off state. All of this is written in the main loop, which repetitively performs these actions.

Unfortunately, delay statements block any other things happening in the sketch during the delay, which become increasingly obvious as you write more complicated sketches. The LedFlasher library blinks LEDs without blocking the rest of the sketch by reading the value of the microprocessor's clock. It looks to see how much time has elapsed since the previous time an LED was turned on using the millis() function. It's a standard C++ function—the Arduino IDE is basically C++ with some Arduino specific functions added in. So, if a set number of milliseconds have passed, the LED turns off, and then the LedFlasher instance waits for a set number of milliseconds before it turns the LED back on. *NOTE: I don't really know how arduino-timer works; I think it digs into the core Arduino libraries to do its magic.*

The Arduino-timer, however, can be used to time anything in your code. And the timer can run once, run repetitively or run at a certain time. You can create multiple timers and multiple tasks to each timer. In this sketch, I'm using the timer to fade up and down the deflector glow and the warp nacelle glow.

Like the LedFlasher, first an instance of a timer is created and then assigned tasks. There are two ways to create an instance, and this sketch uses the simpler, more familiar method:

```
auto sysTmr = timer_create_default();
```

A task is assigned in the `setup` block that will fade up the deflector glow:

```
sysTmr.every(fadeIncr, fader, "fade up");
```

This task will repetitively call the `fader` function using the time value stored in the variable `fadeIncr`. It will send the argument or parameter `"fade up"` to the function after the specified time. Also, notice this happens immediately after a `delay` that is meant to sync it to the spacedock startup sequence in the first two Star Trek movies. In this case, code is being intentionally blocked to prevent anything from happening.

warpBtnFlag

Subsequent fade downs and fade ups are triggered when a momentary button is pressed. In the main loop, you can the if statement that detects the button press:

```
if (digitalRead(WARP_BTN_PIN) == HIGH && !warpBtnFlag) {
  sysTmr.every(fadeIncr, fader, "fade down");
  warpBtnFlag = 1;
}
```

There is a flag—`warpBtnFlag`—that makes sure the task isn't reassigned until a complete

fade down/up cycle is completed.

We'll examine the fader function at the end of the sketch.

ATtiny85 sketch

```
/*
Flashing collision strobes and navigation lights;
changing deflector glow from amber to blue;
ramping nacelle glow from dark to blue;
on Enterprise refit model using ATtiny85
Inspired by Ostrich Longneck
LedFlasher library by Nick Gammon
Arduino-timer by Michael Contreras
Jennifer Petkus
March 24, 2021
*/
#include <LedFlasher.h> // NOTE: there's no semi-colon at the end of the include directive
#include <arduino-timer.h>

LedFlasher strobeLights (2, 900, 100); // NOTE! Pin numbers relate to input or output
numbers, not physical layout of ATtiny
LedFlasher navLights (3, 500, 1000);

const int IMP_PIN = 0;
const int WARP_PIN = 1;
const int WARP_BTN_PIN = 4; // connects to button that switches from impulse to warp
bool warpFlag = 0; // 0: at impulse; 1: at warp
bool warpBtnFlag = 0; // 0: deflector not changing; 1: deflector changing
int fadeVal = 0;

// first number is how long fade takes, then divided by PWM values
unsigned long fadeIncr = 2000 / 255;
// don't make first number too long or steps will be obvious

// adjust this to match drydock spotlight startup sequence
unsigned long startupDelay = 79299 + 4000;
// startup sequence starts over when ATtiny85 and ATMega328 are reset

auto sysTmr = timer_create_default();

void setup() {
  strobeLights.begin();
  navLights.begin();
  pinMode(IMP_PIN, OUTPUT);
  pinMode(WARP_PIN, OUTPUT);
  pinMode(WARP_BTN_PIN, INPUT);
  analogWrite(IMP_PIN, 0);
  analogWrite(WARP_PIN, 0);
  warpFlag = 1; // we want be at impulse on startup so warpFlag is set
  delay(startupDelay);
  sysTmr.every(fadeIncr, fader, "fade up"); // initial fade up to amber
}

void loop() {
  sysTmr.tick();
  strobeLights.update();
  navLights.update();
  // warpBtnFlag prevents overlapping imp/warp state change
  if (digitalRead(WARP_BTN_PIN) == HIGH && !warpBtnFlag) {
```

```
    sysTmr.every(fadeIncr, fader, “fade down”);
    warpBtnFlag = 1;
  }
}

bool fader(const char *which) {
  int down_pin, up_pin;
  if (warpFlag) { // if at warp fade nacelle down and impulse up
    down_pin = WARP_PIN;
    up_pin = IMP_PIN;
  } else { // if at impulse fade impulse down and nacelle up
    down_pin = IMP_PIN;
    up_pin = WARP_PIN;
  }
  //fade down then up
  if (which == “fade down”) { // fade down
    fadeVal--;
    if (fadeVal <= 0) {
      sysTmr.cancel();
      analogWrite(down_pin, 0);
      sysTmr.every(fadeIncr, fader, “fade up”);
    } else {
      analogWrite(down_pin, fadeVal);
    }
    return true;
  } else if (which == “fade up”) { // fade up
    fadeVal++;
    if (fadeVal >= 255) {
      sysTmr.cancel();
      warpBtnFlag = 0;
      warpFlag = !warpFlag;
      return false;
    } else {
      analogWrite(up_pin, fadeVal);
      return true;
    }
  }
}
```

fader( ) function

Functions are used when you need to do the same thing multiple times in a sketch. If you ever find yourself writing the same code more than once, it’s best to write a function. Functions also reduce the length of a `loop` block, making it less cluttered. Functions are usually specific to the sketch you’re writing. Classes are ways of making functions easily accessible to more than one sketch. And libraries are convenient ways of making these classes accessible to a larger number of people. As already noted, we’re using two libraries here and these libraries are made up of classes. We declared instances of these classes and then use the functions or methods defined in these classes.

In this sketch, it would be possible, especially in this simple sketch, to include everything in the fader function directly into the main loop block … if we weren’t using arduino-timer tasks. These tasks have to call a function when the task completes.

Let’s look at how the function works. The function describes a cycle. Everytime the warp button is pressed, it assigns a task that will send the `“fade down”` argument or parameter

to the function. When the function is called, it looks to see what parameter has been sent …

```
if (strcmp(which, “fade down”) == 0) {
```

… and acts accordingly. The appropriate pin of the ATtiny is slowly faded until it reaches zero …

```
if (fadeVal <= 0) {
```

… at which point the the current task is canceled and a new task assigned that sends the parameter `“fade up”` to the `fader()` function.

```
sysTmr.cancel();
analogWrite(down_pin, 0);
sysTmr.every(fadeIncr, fader, “fade up”);
```

You might notice in the line that defines the fader function …

```
bool fader(const char *which) {
```

… that any parameter sent to the `fader()` function is assigned to the local variable (only valid within the function) `which`, and that we use a string compare function (a core C++ function) to see whether which contains `“fade up”` or “fade down.”

warpFlag

Also notice that the very beginning of the `fader()` function decides which pins are being faded up or down based on `warpFlag`. Be aware that `warpFlag` is different from `warpBtnFlag`, although they are both Boolean variables. `warpBtnFlag` is used to prevent overlapping button presses. `warpFlag`, however, keeps track of whether the ship is at impulse speed or warp. The variable was set to `false` when it was declared at the top of the sketch—you’re not traveling at warp in space dock. When a fade down/up cycle is completed, we change the value of `warpFlag` to the opposite of `warpFlag`:

```
warpBtnFlag = 0;
warpFlag = !warpFlag;
```

And we also set `warpBtnFlag = 0`, which means button presses are again recognized.

Finally, I should point out the various `return true;` and `return false;` statements sprinkled throughout the function. arduino-timer tasks expect that the function it calls returns either true or false to confirm whether the task should repeat or stop. So after this statement:

```
sysTmr.every(fadeIncr, fader, “fade up”);
```

The task expects a `true` value to be returned. If we cancel a task and don’t reassign it, for instance after the fade down has completed, we return `false`. The Boolean return values are also the reason for the `bool` keyboard before the declaration of the fader function. The other functions in the sketch—setup and loop—don’t return any values and thus the `void` keyword in their declarations.

More things to note

Let's look at a couple of things I should elaborate on:

Timer variables

You might notice these variable declarations:

```
unsigned long fadeIncr = 2000 / 255;
unsigned long startupDelay = 1000;
```

You're probably more familiar with the `int` or `float` keywords when declaring a variable. `int`s are variables that can hold any whole number between -32,768 to 32,767, but unfortunately that's not a very long period of time counted in milliseconds, which is the default value arduino-timer uses to measure intervals. (The alternate method of creating a timer instance allows microsecond time values.) An `unsigned long` variable, on the other hand, is a whole number between 0 and 4,294,967,295, which works out to a long time in milliseconds, and you're unlikely to see that number reached in any sketch you write. It's probably overkill, but in general it's best to write any values used in timer functions as `unsigned long` variables.

Tick, tick, tick

Also notice the line …

```
sysTmr.tick();
```

… at the start of the `loop` block. This line just tells any arduino-timer tasks to see if it's time to do something and is equivalent to the `update()` call used by LedFlasher.

Function parameters

Another peculiarity that might surprise some is the way the parameter in the fader function is determined. Coming from less strictly typed programming languages such as JavaScript, php and ActionScript 3.0, someone would be tempted to write:

```
if (which == "fade down") {
```

But this construction is problematic, so I've been told. In fact, strings in general are problematic in C++. Whether a `strcmp` is necessary or not, I'm uncertain, but it certainly works, so I'll stick with it.

startupDelay

Notice the line:

```
unsigned long startupDelay = 79299 + 4000;
```

… way back at the top of the sketch. As I mentioned, there's no communication between the sketch running on the ATMega328 and the sketch on the ATtiny85. The ATMega sketch has a long spacedock startup sequence that takes about two minutes to complete. About 80 seconds into that sequence, the main deflector is supposed to fade up to amber, just about when the last spotlight shines on the engineering hull registry. But the ATMega sketch doesn't control the deflector dish, the ATtiny sketch does. That's why there's a delay

here to make the fade up of the deflector match the activation of the spotlight.

Unfortunately, the ATtiny starts up a lot faster than the ATMega and that's why there's an additional four seconds added here to adjust for that. You'll have to tweak this value if your ATMega has a different startup delay.

Oh, the secondary purpose this delay provides is that it prevents going to warp while in spacedock, but the spacedock musical sequence actually lasts 1:48 seconds. After 83.299 seconds, you could go to warp—while in spacedock. To fix this, instead of a delay, you could use a timer task to stage the activation of the collision strobes, navigation light and deflector, and another task timed to remove a flag that prevents the warp button from working until the spacedock startup sequence completes. The process would be similar to the `startUp` function in the next sketch.

Finally

In conclusion, you can see that this sketch is pretty simple and it does work without a hitch. You might be wondering, however, whether it was necessary to use two libraries. The functionality of the LedFlasher library could have been accomplished with additional tasks assigned to `sysTmr`, but I really love the simplicity of LedFlasher.

After all, the library accomplishes the bare minimum lighting of most starships—collision strobes and navigation lights. It would be perfectly adequate for lighting the Defiant from Deep Space Nine, for instance, or even the Enterprise refit if you didn't care about weapons fire and didn't need to switch from impulse to warp.

If you decide to dispense with LedFlasher, you'll need to create a function similar to fader that blinks these lights, so while in some ways it might seem simpler to just use the arduino-timer library, it actually becomes a little more complicated.

spotlights_rcs_weapons_sound_ATMega.ino

This sketch is more complicated than the ATtiny sketch and it introduces a third library, used to control the Adafruit FX Sound Board, and a fourth library, SoftwareSerial. First, let's talk about the sound board. The Adafruit_Soundboard.h library was downloaded, extracted and placed in the libraries folder inside the Arduino folder that on my Mac was installed in Documents.

< Adafruit_Soundboard

We communicate with the sound board using serial read and write, normally accomplished with pins 1 and 0, which incidentally are the same pins used to communicate with the Arduino Uno. You'll see the LEDs for the RX and TX flash rapidly when uploading a script. The Adafruit example scripts, however, use SoftwareSerial to communicate with the board. SoftwareSerial is one of the "built-in" libraries that was installed with the Arduino IDE and it allows you to use any of the pins (there may be restrictions, but I don't know what they are) for serial communication. These two lines …

```
SoftwareSerial ss = SoftwareSerial(SFX_TX, SFX_RX);
Adafruit_Soundboard sfx = Adafruit_Soundboard(&ss, NULL, SFX_RST);
```

… create an instance of the SoftwareSerial class, using the variables we previously defined …

```
#define SFX_TX 13
#define SFX_RX 12
#define SFX_RST 11
```

Note: I don't know why these are created with #define directives instead of just int. Nor do I know why the `#include "Adafruit_Soundboard.h"` *uses quotes instead of < and >. These are just mysteries of too little consequence for me to investigate. I'm content to follow the lead of the Adafruit example sketches.*

If you want to use pins 0 and 1, which I use to fire phasers, then you'd remove the line where the SoftwareSerial instance is created and you'd change the next line to:

```
Adafruit_Soundboard sfx = Adafruit_Soundboard(&Serial1, NULL, SFX_RST);
```

This tells the instance of the Adafruit_Soundboard class to use the regular hardware serial pins. Of course, you'd then have to change the other pin assignments and the perboard layout in this book would no longer be valid. But you could if you really wanted … just saying. Well, I hope I've convinced you to stick with SoftwareSerial. There must be some reason it's the default choice.

Later on in the script, we'll be using the `play()` and `stop()` methods of our sound board instance—`sfx`—to play and stop sounds. We'll also read the value of the `SND_DET` pin to know whether a sound a playing. You don't want to stop a sound if that wasn't playing—it really causes problems.

< Tasks

A difference you'll notice here compared to the previous sketch is that we're using a differ-

ent way to create an instance of the arduino-timer class and explicity creating tasks:

```
Timer<4> sysTmr;
Timer<>::Task rcsTsk;
Timer<>::Task phasTsk;
Timer<>::Task torpTsk;
Timer<>::Task spotTsk;
```

The instance is still called `sysTmr`, but this statement—`Timer<4>`— specifies a maximum of four tasks. The tasks are created with this constructor: `Timer<>::Task`

The tasks are also handled here quite differently than the previous sketch. The `spotTsk` is fist assigned at the end of the setup block …

```
startDelay = millis();
spotTsk = sysTmr.at(SPOT_1_TIME + startDelay, startUp, “spot1”);
```

… and is then sequentially reassigned throughout the startup sequence, lighting up spotlights 2–7 and in its last act, assigning `rcsTsk`. Note the line above that sets the `startDelay` variable to the current millisecond since the start of the sketch. This is necessary because there's a delay before the ###INTROWAV file starts playing. The `startDelay` is added to keep the spotlights in time with the music.

ATMega sketch

```
#include <arduino-timer.h>
#include <LedFlasher.h>
#include <SoftwareSerial.h>
#include “Adafruit_Soundboard.h”

// detect if ATtiny has gone to warp
const int WARP_DET = 19;

// pins to 74HC595
const int DATA_PIN = 18;
const int LTCH_PIN = 17;
const int CLK_PIN = 16;

// detect if sound still playing
const int SND_DET = 15;

// detect weapons fire
const int WEAP_BTN = 14;

// pins to soundboard
#define SFX_TX 13 // connects to TX on soundboard
#define SFX_RX 12 // connects to RX on soundboard
#define SFX_RST 11 // connects to RST on soundboard

// torpedoes
const int TORP1_PIN = 10;
const int TORP2_PIN = 9;

// spotlights
const int SPOT_1 = 8;
const int SPOT_2 = 7;
const int SPOT_3 = 6;
const int SPOT_4 = 5;
```

```
const int SPOT_5 = 4;
const int SPOT_6 = 3;
const int SPOT_7 = 2;

// phasers
const int PHAS_PIN1 = 1;
const int PHAS_PIN2 = 0;

SoftwareSerial ss = SoftwareSerial(SFX_TX, SFX_RX);
Adafruit_Soundboard sfx = Adafruit_Soundboard(&ss, NULL, SFX_RST);

/* Next value is a very loose approximation of the time
for a torpedo launch burst. Torpedo brightness increases
until it reaches threshold value, goes to 255, and then fades */
const unsigned long TORP_DUR = 2000/255;
const int TORP_THRESH = 64; // when torp launcher switches to full bright
int torpInt = 0; // current brightness of torp launcher
int torpCnt = 0; // port or starboard launcher
int fireCnt = 0; // how many times weapons fired
int fireLmt = 0; // max number weapons fire
const int FIRE_LOW = 2;
const int FIRE_HIGH = 7;
bool weapFlag = 0; // prevents overlapping fire
bool warpFlag = 0; // prevents RCS firing at warp
bool phas1Flag = 0;
bool phas2Flag = 0;
const int PHAS_LOW = 500;
const int PHAS_HIGH = 1500;
unsigned long startDelay = 0;

bool startupFlag = 1; // prevents weapons fire while in spacedock
const unsigned long SPOT_1_TIME = 69072;
const unsigned long SPOT_2_TIME = 70825;
const unsigned long SPOT_3_TIME = 72306;
const unsigned long SPOT_4_TIME = 74131;
const unsigned long SPOT_5_TIME = 75794;
const unsigned long SPOT_6_TIME = 77636;
const unsigned long SPOT_7_TIME = 79299;
const unsigned long RCS_TIME = 109500;

const int rcsPatterns [10] = {160, 80, 40, 20, 10, 5, 18, 17, 129, 66};
// following two values are used to random generate pauses between RCS bursts
const int RCS_LOW = 5000;
const int RCS_HIGH = 10000;

LedFlasher PHAS_1(PHAS_PIN1, 50, 100); // phaser flicker effect
LedFlasher PHAS_2(PHAS_PIN2, 50, 100); // phaser flicker effect

Timer<4> sysTmr;
Timer<>::Task rcsTsk;
Timer<>::Task phasTsk;
Timer<>::Task torpTsk;
Timer<>::Task spotTsk;

void setup() {
  pinMode(SPOT_1, OUTPUT);
  pinMode(SPOT_2, OUTPUT);
  pinMode(SPOT_3, OUTPUT);
  pinMode(SPOT_4, OUTPUT);
  pinMode(SPOT_5, OUTPUT);
  pinMode(SPOT_6, OUTPUT);
```

```
  pinMode(SPOT_7, OUTPUT);

  digitalWrite(SPOT_1, LOW);
  digitalWrite(SPOT_2, LOW);
  digitalWrite(SPOT_3, LOW);
  digitalWrite(SPOT_4, LOW);
  digitalWrite(SPOT_5, LOW);
  digitalWrite(SPOT_6, LOW);
  digitalWrite(SPOT_7, LOW);

  pinMode(WARP_DET, INPUT);
  pinMode(TORP1_PIN, OUTPUT);
  pinMode(TORP2_PIN, OUTPUT);
  pinMode(WEAP_BTN, INPUT);
  pinMode(SND_DET, INPUT);
  pinMode(LTCH_PIN, OUTPUT);
  pinMode(DATA_PIN, OUTPUT);
  pinMode(CLK_PIN, OUTPUT);

  PHAS_1.begin();
  PHAS_2.begin();

  ss.begin(9600);
  sfx.reset(); // essential, if not here, sound will not start
  sfx.playTrack("###INTROOGG\n");
  while (digitalRead(SND_DET) == HIGH) {
  }
  // it takes some time before the sound begins playing and we capture that time below
  startDelay = millis();
  // and add that time to all our spotTsk assignments
  spotTsk = sysTmr.at(SPOT_1_TIME + startDelay, startUp, "spot1");
  digitalWrite(LTCH_PIN, LOW);
  shiftOut (DATA_PIN, CLK_PIN, MSBFIRST, 0);
  digitalWrite(LTCH_PIN, HIGH);
}

void loop() {
  sysTmr.tick();
   if (digitalRead(WEAP_BTN) == HIGH && !weapFlag && !startupFlag) { // prevents
overlapping fire and firing in spacedock
    fireLmt = random(FIRE_LOW,FIRE_HIGH); // max number weapons fire
    weapFlag = 1;
    randomFire();
  }
  if (analogRead(WARP_DET) > 800 && warpFlag == 0) { // prevents RCS from firing at warp
    warpFlag = 1;
    warpMode("warp");
  } else if (analogRead(WARP_DET) < 600 && warpFlag == 1) {
    warpFlag = 0;
    warpMode("impulse");
  }
  if (phas1Flag) {
    PHAS_1.update();
  }
  if (phas2Flag) {
    PHAS_2.update();
  }
}

void warpMode(const char *which) {
  if (strcmp(which, "warp") == 0) {
```

```
    if (digitalRead(SND_DET) == LOW) {
      sfx.stop();
    }
    sfx.playTrack("####WARPWAV\n");
  } else if (strcmp(which, "impulse") == 0) {
    if (digitalRead(SND_DET) == LOW) {
      sfx.stop();
    }
    sfx.playTrack("#IMPULSEWAV\n");
    rcsTsk = sysTmr.in(random(RCS_LOW, RCS_HIGH), fireRCS, "trigger");
  }
  delay(4000);
  if (digitalRead(SND_DET) == LOW) {
    sfx.stop();
  }
   sfx.playTrack("#AMBIENTOGG\n");
}

bool startUp(const char *which) {
  if (strcmp(which, "spot1") == 0) {
    spotTsk = sysTmr.at(SPOT_2_TIME + startDelay, startUp, "spot2");
    digitalWrite(SPOT_1, HIGH);
  }
  if (strcmp(which, "spot2") == 0) {
    spotTsk = sysTmr.at(SPOT_3_TIME + startDelay, startUp, "spot3");
    digitalWrite(SPOT_2, HIGH);
  }
  if (strcmp(which, "spot3") == 0) {
    spotTsk = sysTmr.at(SPOT_4_TIME + startDelay, startUp, "spot4");
    digitalWrite(SPOT_3, HIGH);
  }
  if (strcmp(which, "spot4") == 0) {
    spotTsk = sysTmr.at(SPOT_5_TIME + startDelay, startUp, "spot5");
    digitalWrite(SPOT_4, HIGH);
  }
  if (strcmp(which, "spot5") == 0) {
    spotTsk = sysTmr.at(SPOT_6_TIME + startDelay, startUp, "spot6");
    digitalWrite(SPOT_5, HIGH);
  }
  if (strcmp(which, "spot6") == 0) {
    spotTsk = sysTmr.at(SPOT_7_TIME + startDelay, startUp, "spot7");
    digitalWrite(SPOT_6, HIGH);
  }
  if (strcmp(which, "spot7") == 0) {
    digitalWrite(SPOT_7, HIGH);
    spotTsk = sysTmr.at(RCS_TIME + startDelay, startUp, "rcs");
  }
  if (strcmp(which, "rcs") == 0) {
    startupFlag = 0;
    while (digitalRead(SND_DET) == LOW) {
    }
    sfx.playTrack("#AMBIENTOGG\n");
     rcsTsk = sysTmr.in(random(RCS_LOW, RCS_HIGH), fireRCS, "trigger");
  }
  return false;
}

void randomFire() {
  if (digitalRead(SND_DET) == LOW) {
    sfx.stop();
  }
```

```
  if (fireCnt < fireLmt) {
    if (random(0,2) == 1) {
      phasTsk = sysTmr.in(100, firePhaser, “fire”);
    } else {
      torpTsk = sysTmr.in(100, fireTorpedo, “ramp”);
    }
    fireCnt++;
  } else {
    fireCnt = 0;
    weapFlag = 0;
    sfx.playTrack(“#AMBIENTOGG\n”);
  }
}

bool fireTorpedo(const char *which) {
  if (strcmp(which, “ramp”) == 0) { // slowly lights LED until …
    if (torpInt > TORP_THRESH) { // … flash after threshold
      if (torpCnt == 0) {
        sfx.playTrack(“#TORPEDOWAV\n”);
      }
      torpInt = 255;
      torpTsk = sysTmr.in(TORP_DUR/2, fireTorpedo, “fade”);
    } else {
      torpTsk = sysTmr.in(TORP_DUR, fireTorpedo, “ramp”);
      torpInt++;
    }
  } else {
    if (torpInt <= 0) {
      if (torpCnt == 0) {
        torpTsk = sysTmr.in(TORP_DUR, fireTorpedo, “ramp”);
        torpCnt = 1;
        torpInt = 0;
      } else {
        torpCnt = 0;
        torpInt = 0;
        randomFire();
      }
    } else {
      torpTsk = sysTmr.in(TORP_DUR/2, fireTorpedo, “fade”);
      torpInt--;
    }
  }
  int cur_pin = 0;
  if (torpCnt == 0) { // switches between port / stbd launcher
    cur_pin = TORP1_PIN;
  } else {
    cur_pin = TORP2_PIN;
  }
  analogWrite(cur_pin, torpInt);
  return false;
}

bool firePhaser(const char *which) {
  if (random(0, 2) == 0) {
    phas1Flag = 1;
  } else {
    phas2Flag = 1;
  }
  if (strcmp(which, “fire”) == 0) {
    sfx.playTrack(“##PHASERWAV\n”);
    phasTsk = sysTmr.in(random(250,750), firePhaser, “stop”);
```

```
  } else {
    sfx.stop();
    digitalWrite(PHAS_PIN1, LOW);
    digitalWrite(PHAS_PIN2, LOW);
    phas1Flag = 0;
    phas2Flag = 0;
    randomFire();
  }
  return false;
}

bool fireRCS(const char *which) {
  if (warpFlag == 0) {
    if (strcmp(which, "trigger") == 0) {
      int display = rcsPatterns[random(0,10)];
      digitalWrite(LTCH_PIN, LOW);
      shiftOut (DATA_PIN, CLK_PIN, MSBFIRST, display);
      digitalWrite(LTCH_PIN, HIGH);
      rcsTsk = sysTmr.in(200, fireRCS, "cancel");
    } else {
      digitalWrite(LTCH_PIN, LOW);
      shiftOut (DATA_PIN, CLK_PIN, MSBFIRST, 0);
      digitalWrite(LTCH_PIN, HIGH);
      rcsTsk = sysTmr.in(random(RCS_LOW, RCS_HIGH), fireRCS, "trigger");
    }
  }
  return false;
}
```

startUp( ) function

The `startUp` function is repetitively called by `spotTsk`, each time with a different parameter, that goes from "spot1" to "spot7" and finally "rcs." With each call to the function, a different spotlight is lit. With the final call, the `spotTsk` sets up the `rcsTsk` to call the `fireRCS` function.

warpMode( ) function

This function is called when pin A5 (numbered pin 28) on the ATMega detects pin 1 (numbered pin 6) on the ATtiny is outputting a value greater than 800 or less than 600. Remember, the ATtiny pin is outputting PWM values from 0 to 1023. These values are somewhat arbitrary and you may have to fiddle with these values. The analog input pin on the ATMega we're using has trouble reading PWM values, that's why we're bridging a 100µF electrolytic capacitor from this pin to ground.

It's still a little tricky to read the value of the ATtiny pin, and we're aided by using `warpFlag` …

```
if (analogRead(WARP_DET) > 800 && warpFlag == 0) {
```

… to prevent multiple calls to the warpMode function. There's also a `delay()` in the function that prevents any other actions occurring while the appropriate audio file is playing. Because our strobes and nav lights are being driven by the ATtiny, we don't really notice the blocking delay.

randomFire( ) function

All the pins on the ATMega and ATtiny are used. On the ATMega, that means there's only pin connected to a fire weapons button, not separate buttons for fire phasers and torpedoes. The randomFire function fires either phasers or torpedoes as many as six times but no less than two, as specified at the start of the sketch:

```
const int FIRE_LOW = 2;
const int FIRE_HIGH = 7;
```

and in the main loop where the weapons fire button press is detected:

```
if (digitalRead(WEAP_BTN) == HIGH && !weapFlag && !startupFlag) { // prevents overlapping
fire and firing in spacedock
  fireLmt = random(FIRE_LOW,FIRE_HIGH); // max number weapons fire
  weapFlag = 1;
  randomFire();
}
```

`fireLmt` is set between the values in `FIRE_LOW` and `FIRE_HIGH`. Each time the `randomFire` function is called it flips a coin whether to fire phasers or torpedoes. Both port and starboard torpedoes fire, counting just once in the variable `fireCnt`. There are two pins on the ATMega that fire different phaser LEDs. You might wire one of the pins to a phaser cannon on the top of the saucer and one underneath or on the engineering hull. The function will clip a coin which phaser to fire. You might be able to wire two LEDs to each pin to make two phaser cannons light up at a time. It might overtax the current draw of the ATMega, however. If that happens, you could direct the output of the pins to a transistor, just like we do with the collision strobes and navigation lights on the ATtiny.

If the `randomFire()` function decides to fire phasers, it calls the `firePhaser()` function, otherwise it calls the `fireTorpedo()` function.

When the `fireCnt` exceeds `fireLmt`, the `randomFire()` function ends after setting the `weapFlag` to zero, making the weapons button active again.

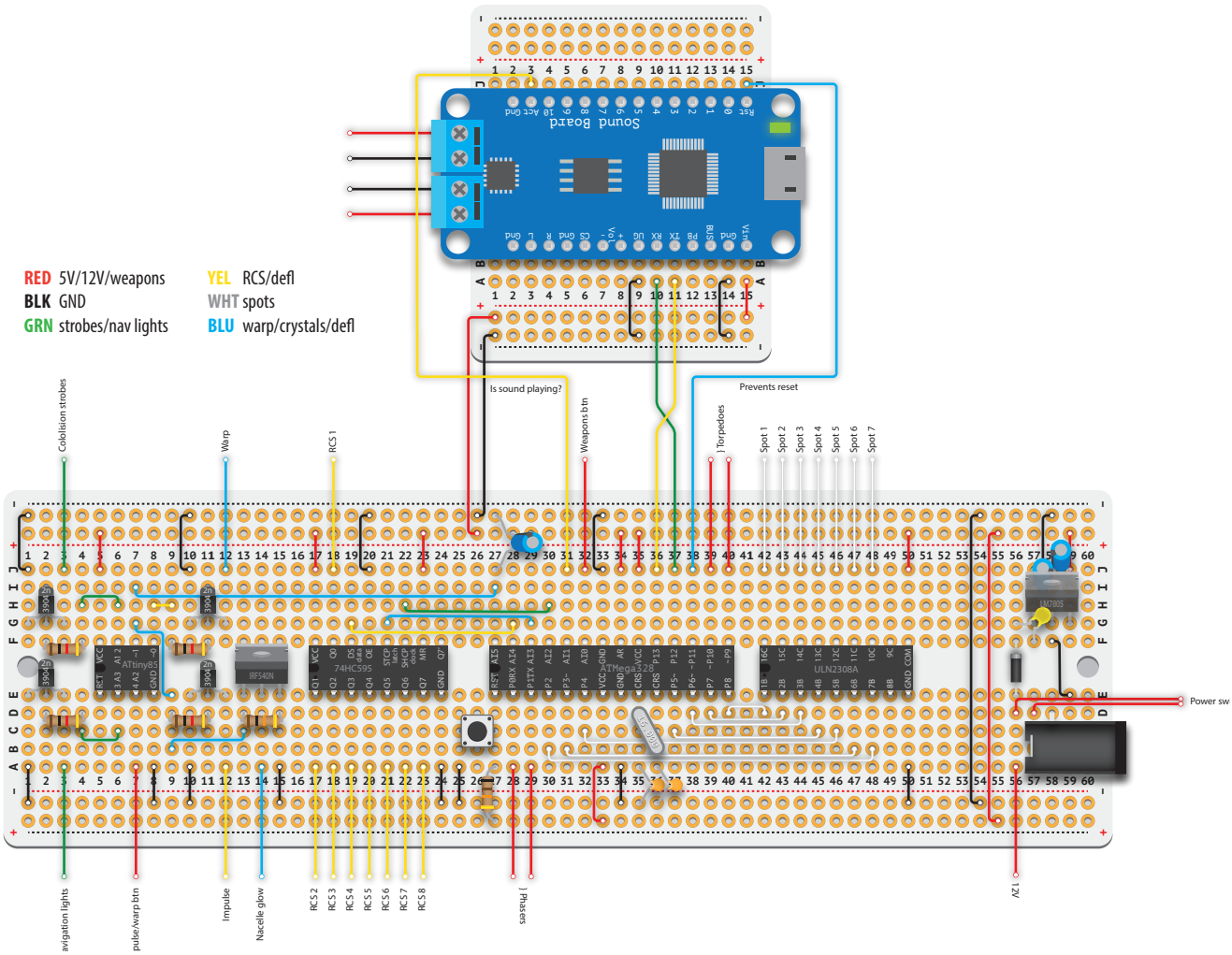
firePhaser( ) function

This function is pretty simple. It rolls the dice as to which phaser to fire and then sets a flag, either `phas1Flag` or `phas2Flag`, to high. Back in the main loop, there are two `if` statements that check to see whether either of these flags are set. If one is, it sends an update to an instance of the LedFlasher that corresponds to that phaser.

Remember that the `main` loop is constantly running. In the ATtiny sketch, we update the LedFlasher instance for the strobes and nav loops every time the main loop runs. In the ATMega sketch, we only want to update the phaser instances for the short period the phasers are firing. Notice also that in the phasTsk assignment …

```
phasTsk = sysTmr.in(random(250,750), firePhaser, "stop");
```

… we're doing another `random()` to vary the time the phaser fires from a quarter to three-quarters of a second. We're also playing the phaser sound effect and stopping it



Examining the board(s)

Woof, that's daunting! These two boards contain the essentials of the starship lighting/audio circuit. You'll notice the complete absence of LEDs because those are scattered throught the Enterprise refit model. Also there are no buttons on the boards because they will be mounted on the control panel of the display stand (see below).

This board is designed to sit outside the model, probably tucked away in the display stand.[1] Most of the wires you see dangling off the edge of the board need to enter the ship, probably through the use of Dupont header pins. There are two sets of wires that connect to the impulse/warp button and the weapons fire button. There's another button that connects to the reset pins of the ATtiny85 and the ATMega328. There's a button that is the switch that connects the 12V power supply with the 5V voltage regulator. And one wire provides 12V power to the LED strips that provide the window lighting.

I have more or less followed the color code displayed above. Any pin that connects to

1 You might be able to fit these board inside the saucer section, but it would make lighting the windows difficult. It would be easier to distribute the circuit across a number of quarter perfboards.

voltage is connected with red wire; to ground black wire; to any of the blinking lights green; the the RCS thrusters or the deflector dish yellow; to the spotlights white; and to anything indicating warp mode, including the deflector, blue.

Sound board

Let's start with the quarter perfboard that holds the Adafruit FX Sound Board. As you can see it takes up most of the board. You could solder directly to the holes on the sound board, unless you opt to solder the supplied male header pins into those holes to make it easier to use in a breadboard. In which case you would solder onto the holes of the perfboard.

The sound board takes its power from the rails of the main board. The TX, RX, Act and Rst pins of the sound board are connected to the ATMega. Notice also that UG connects to GND, which puts the board into serial instead of trigger control. Finally, the two terminal blocks connect to speakers.

Main board

The board is roughly divided into three areas, starting from the left up to column 15 with the ATtiny85, four transistors and a MOSFET, all running the strobes_navs_deflector_on_ATtiny.ino sketch. From column 16–50, you'll find the 74HC595, the ATMega328 and the ULN2803A. From column 56–60, you'll find the LM7805 voltage regulator, two capacitors, a barrel jack, diode and resettable fuse.

For illustration purposes, the wiring connecting components are shown sitting on the board, but it would be a lot tidier to run those connections on the back side. The perfboard shown is not a double-sided board and anything soldered into the back side connects to the front.

All the chips on the board are facing left and as much as possible, I've tried to make all the components face the same way. Also notice the top and bottom power rails are bridged at columns 54 and 55.

Strobes, navs and deflector

You'll notice four transistors surrounding the ATtiny. The first two on the left connect to the pins that power the collision strobes and navigation lights. The flat face of the transistor is facing us. There are 1KΩ pull-down resistors between the middle base leg of the transistor and the pins on the chip. These transistors should have no problem powering the four strobes and the nine navigation/formation lights.

The second pair of transistors on the right of the chip connect to the impulse and warp pins. If you only cared about lighting the deflector, you could dispense with these transistors and instead connect the impulse and warp pins to an amber/blue bi-color LED. However, we want the impulse pin of the ATtiny to provide power to the amber glow of the deflector, the amber glow of the impulse deflection crystal (the round dome aft of

the bridge) and the red glow of the impulse engine. Meanwhile, the warp pin will provide power to the blue glow of the deflector, the blue glow of the magnatomic amplification crystal (the smaller dome on the top and towards the front of the nacelles) and, of course, the blue glow of the nacelles.

That's too much for a single pin on the ATtiny, so instead we need the transistors to switch the various LEDs to the main 5V supply. Unfortunately, this means the bi-color LED no longer works because the signal coming from the transistors is a connection to ground, whereas the amber and blue pins on the bi-color LED are expecting positive voltage. There are ways around this, but the simplest solution is to just use separate amber and blue LEDs.

The output of the warp pin on the ATtiny is also connected to the IRF540N MOSFET. The middle or drain leg of the MOSFET is connected to the ground pins of the LED strip in each nacelle. The positive pin of the LED strip is connected to the 12V supply.

Spotlights, weapons, RCS thrusters, sound

The ATMega is surrounded by the 74HC595 shift register on the left and the ULN2803A Darlington array on the right. In the upper left of the ATMega, you'll see wires connecting it to the clock, latch and data pins of the 595. The 595 controls the firing of the RCS thrusters. Notice the connection to ground of the Output Enable pin and the connection to 5V on the reset pin.

It's easy to miss the wire connecting numbered pin 28 on the ATMega to the warp pin of the ATtiny. This allows the sketch running the ATMega to know when the ship has gone to warp. A little more obvious is the capacitor connecting pin 28 to ground, which smooths out the PWM signal from the ATtiny is into a more readable serial signal.

To the middle of the top half the ATMega, there are several wires connecting to the sound board, as well as power and ground connections to the Mega. You'll also find a wire that connects to the weapons button and wires that connect to the torpedo launchers.

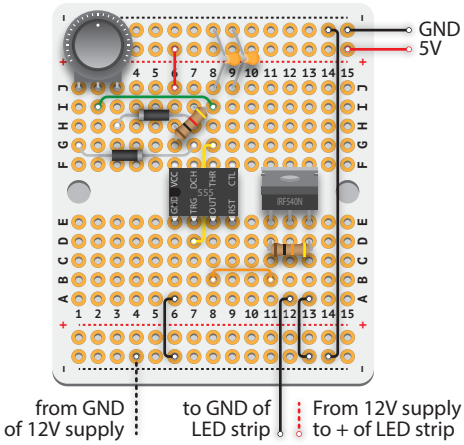
Starting on the bottom left of the ATMega, there's a 10KΩ pull-down resistor connecting to the reset pin and two wires connecting to the phasers. Each of the phaser pins might be able to connect to two LEDs, but to be certain or to light more even more LEDs, it would be best to run the output of these pins to transistors. There is room on the main board between columns 51–55 to put two transistors for this purpose. They would be wired much the same way as the strobe and nav lights on the ATtiny.

Most of the rest of the bottom of the ATMega connects to the Darlington array, numbered pins 4–6 and 11–14. The remaining pins connect to power and ground or read the 16Mhz clock crystal.

Power supply

And finally, the power supply, which takes up only a few columns and is built around the LM7805 voltage regulator. What's not shown is the heatsink attached to the back of the 7805 because that would have obscured the view of the two capacitors. But you can read more about that here.

555 PWM LED strip dimmer



Here's the board that can dim the LED window lighting strips that's mentioned in the Lights and Sound chapter. It only requires a quarter size prototyping board and actually still has room for other uses.

The two diodes used are the same 1N4001 diodes used in the board power supply. Notice how their opposite orientations. It doesn't matter which faces left or right, as long as they differ. Your choice, however, does affect the behavior of the potentiometer, whether a clockwise turn increases brightness or reduces it.

The two ceramic capacitors are both 0.01uF. The resistor across pins 8 and 7 of the 555 is 1KΩ and the one across the gate and source pins of the IRF540N is 10KΩ.

Notice the drain leg is connected to the GND pin of the various LED strips. The V+ pins of the strips is connected to the V+ of the 12V power supply. The GND of the 12V power supply should also be connected to this board, but that's already accomplished if the board is getting its 5V supply from the main board.