Holiday Project

Sequencing and Dimming – Add Some Pizazz to Your Holiday Lights

by Eric Gunnerson

A few years ago, my wife and I accidentally won third place in a holiday light contest in our development. In an attempt to win the rarely coveted first prize, I decided to do a microcontroller-based project. In the years since I first got into electronics (when we — to paraphrase Douglass Adams — thought that digital watches were a neat idea), the amount of equipment needed to do microcontroller work has decreased by quite a bit, so it was a good fit with my first project, an animated Santa, sleigh, and reindeer. In this context, "animated" means lights that are sequenced, not moving parts. At least for this year.

The project discussed in this article is the outgrowth of that first project, and it supports dimming of lights in addition to sequencing them. With two animations in the yard, the general illumination of the house in white lights had become a bit boring, so a change was in order. Rather than use a single string to outline the house, I'll use four (in red, green, blue, and white) and use my dimmer to slowly dim between them. The idea came from somebody who used X-10 dimmers and a computer to do this, but my implementation will be a bit cheaper and will operate stand-alone. It can also do chaser lights and fine (per cycle) control, which I've implemented, but which may not meet aesthetic standards.

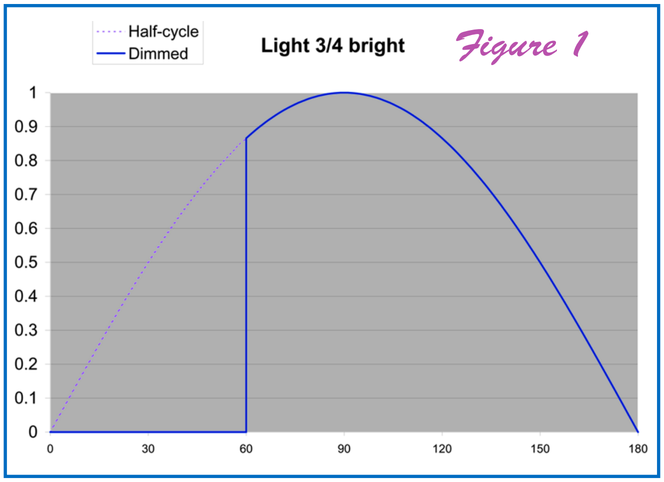
This project involves potentially lethal AC voltages. It's not particularly dangerous, but please keep that in mind when you're working with the AC circuitry. If you use this project outside, you'll need to protect it from the weather.

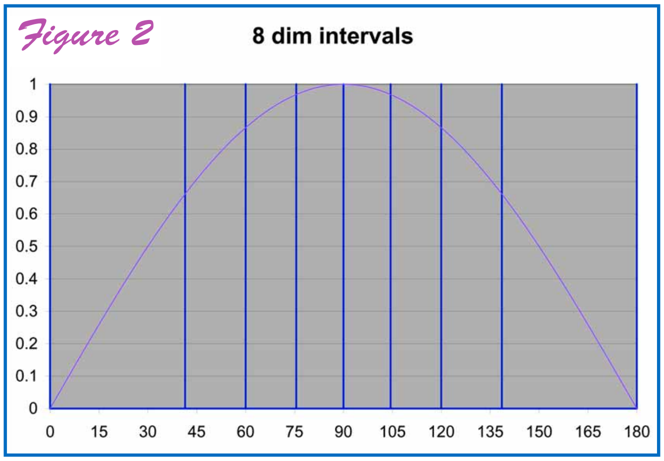
Switching and Dimming AC Lights

There are two ways to dim AC lights. The most obvious one is to simply reduce the voltage, which can be done either with a resistor divider, or an autotransformer (also known as a variac). Neither of these are good solutions in this case, because they aren't easy to control electronically.

Though incandescent light bulbs have very thin filaments, there is still some thermal mass in the filament, as evidenced by the fact that you don't see a 60Hz flash from them. Even though the current is switching on and off 120 times a second, the filament maintains a constant brightness. We can take advantage of this for dimming by controlling the duty cycle of the waveform rather than the voltage of the waveform. To do so requires that we delay our turn-on from the zero-crossing of the waveform.

As you can see in Figure 1, we can get the effect of three-quarters of the voltage by only turning the power on for the appropriate part of the cycle. To get other dim levels, we simply vary the time at which we turn on the power. Since the light output of the bulb depends upon the amount of power we send to it, we need to choose our intervals so that they represent intervals of equal power.





The power is represented by the area under the waveform, which is why the line isn't at the quarter point.

It's easy enough to calculate this for as many dim levels as you want; the important point is that the intervals are not equally spaced. See Figure 2. This approach is fairly common in the DC world, as well; a heater can be nicely "dimmed" by varying

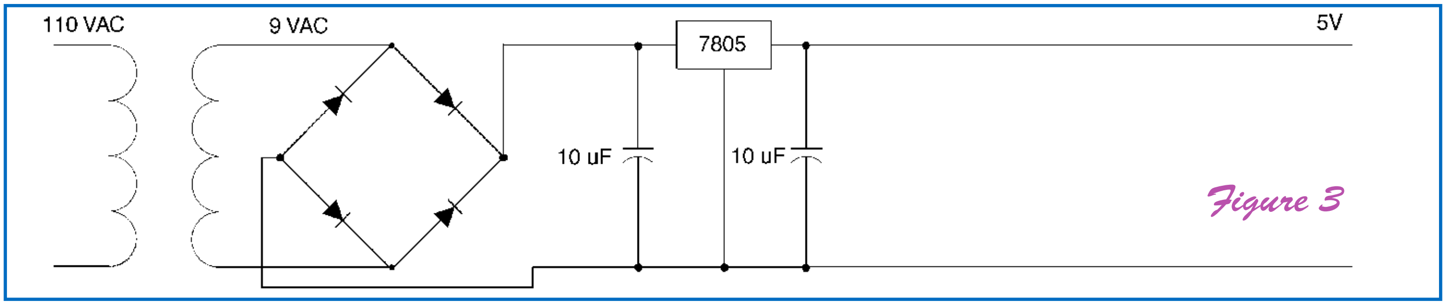


Figure 3

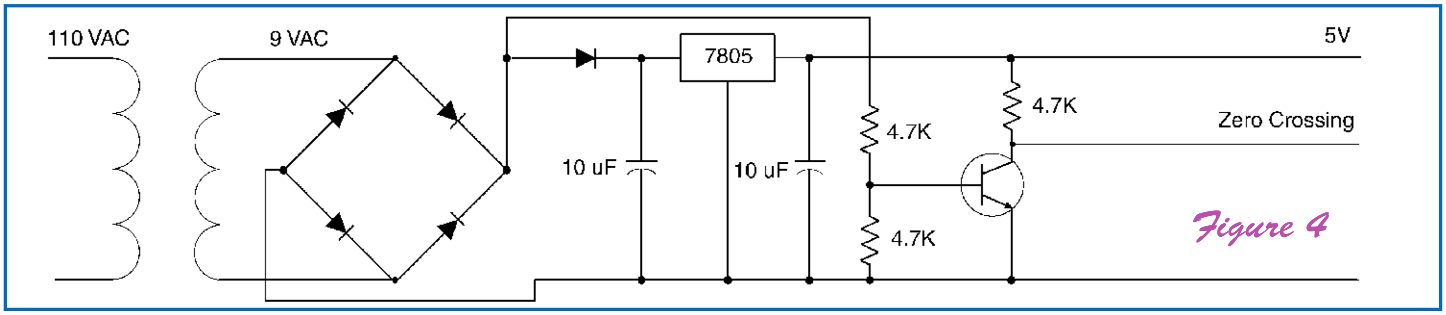
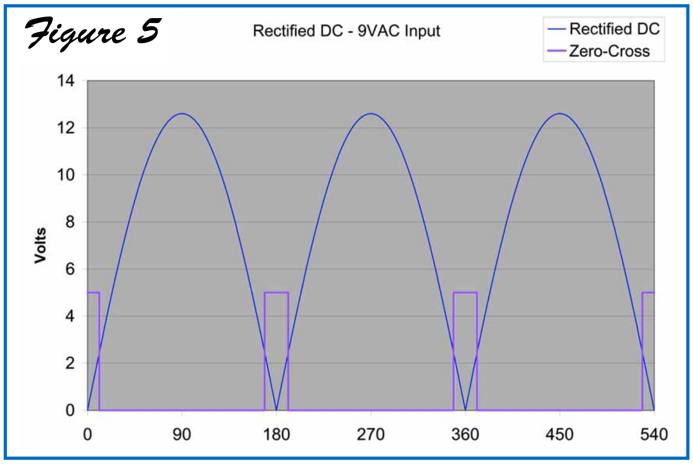


Figure 4

Figure 5



its duty cycle.

Doing the Switching

Now that we know how to do the dimming, we need to figure out how to actually switch the AC circuit. A triac is by far the most popular choice for controlling AC circuits. They are simple to use and interface, but they do have one interesting feature; once you turn a triac on, it doesn't turn off until the voltage across the load goes to zero. This makes them nicely suited to AC control, because the voltage goes to zero each half-cycle.

In most applications, it's easier to use a solid-state relay rather than a triac. A solid-state relay contains a triac, and can be thought of as an AC-switching black box; it isolates the AC world from the DC world (very important for safety), and switches with logic-level inputs. Most are also zero-crossing devices, which means they only switch as the waveform crosses zero, when voltage and current are low. This is nicer on the load (no sudden transients), and generates much less RF noise than switching at a random time.

For dimming, however, we have to be able to switch anywhere in the cycle, so zero-crossing doesn't work. I've chosen to use a solid-state relay without zero-crossing circuitry, though they are less common, and doing this has limited the amount of current I can switch. You can build your own solid-state relays if you wish to control larger amounts of current; see the references for a good starting point.

Hardware and Circuitry

Now that we understand how to do dimming, we need to choose the hardware we'll use to do it. I broke the circuitry into three modules: the zero-crossing circuit/power supply, the microcontroller, and the AC box. All of this will be contained in something sufficiently waterproof; I've found plastic toolboxes a good choice.

Zero-crossing Circuit/Power Supply

The zero-crossing circuit/power supply is quite simple. It starts as a standard 5V power supply using a 7805 regulator and a 9VAC transformer. This will provide power for the microcontroller and the solid-state relays. See Figure 3.

To generate the zero-crossing signal, we need to tap into the power supply after the bridge rectifier, so we get the pulsating DC at this point. The initial filter capacitor's job is to get rid of this signal, so we need to insert a diode in between the bridge rectifier and the capacitor.

Now that we have this signal, we want to generate a pulse when the signal is zero. We add a transistor with pull-up resistor and a voltage divider to the circuit. See Figure 4.

Figure 5 shows the pulsating DC and zero-crossing signals.

As long as the base voltage is greater than the turn-on voltage of the transistor, the transistor will pull the output to ground. When the voltage drops below the turn-on voltage, the transistor will turn off, and the output will be pulled high by the resistor. Since the pull-up is to the 5V supply, we get a nice pulse that is nearly symmetrical around the zero cross point. This signal is connected to an input pin on the microcontroller.

The width of the pulse is determined by the input voltage, the resistors used in the voltage divider, the diode drops in the rectifier, and the voltage at which the transistor turns on. Rather than try to measure this, I put the signal on the scope, zoomed in, and made a reasonable estimate to the width of the signal. This value is used later.

Microcontroller

For this project, I chose a Motorola 68HC11 controller (a 68HC811E2, to be precise). In my earlier projects, I chose this controller because it was fairly inexpensive, easy to deal with, and I had a local resource who had used them, and could provide technical support. It stores its program in EEPROM, and can be programmed over a serial link. It also lets me keep my assembly skills tuned up. For this project, the HC11 is especially nice, because of a feature known as output compare. More about that in the algorithms section.

The HC11 is built using Marvin Green's excellent BotBoard (see references). Though this board was targeted towards robotics, it's perfect for this project because it has a small prototype area for additional circuitry. Construction is very simple as long as you have a fine soldering tip and a magnifier (or young eyes).

AC Box

Putting all the AC components in a separate box makes life a lot easier; you can't shock yourself when working on the BotBoard. I use a standard dual-gang blue plastic box for my AC control. In it live two duplex outlets, giving me four circuits, and the solid-state relays glued to the backs of the outlets. Control signals are carried to the AC box via

Preprinted from December 2000 Nuts & Volts Magazine. All rights reserved. No duplication permitted without permission from T & L Publications, Inc.

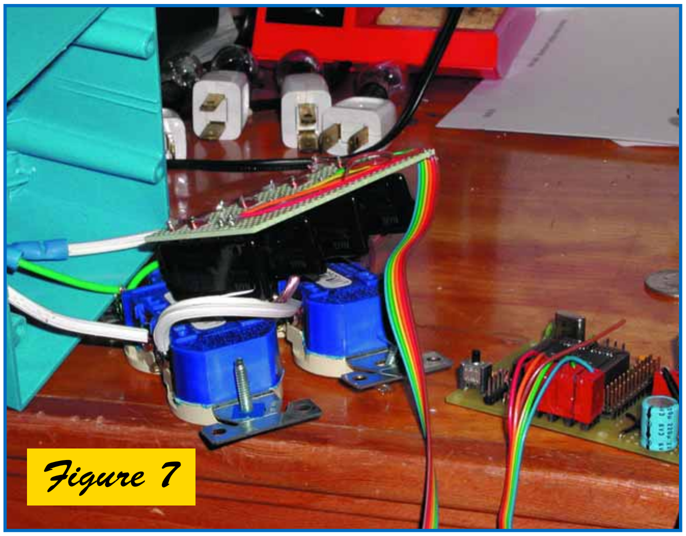


Figure 7

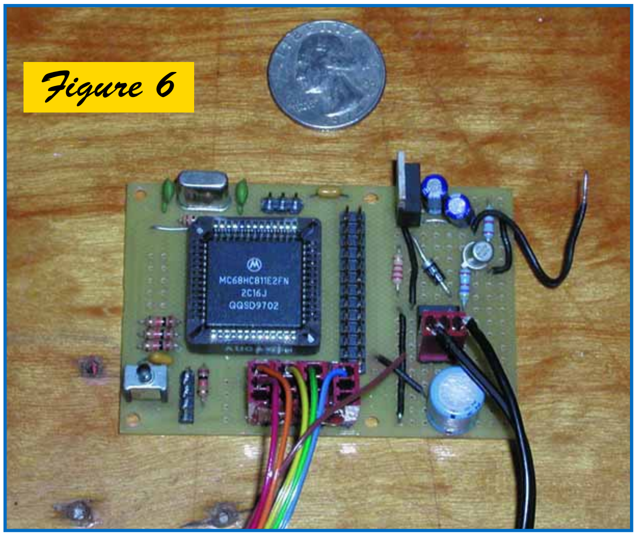


Figure 6

ribbon cable. A 3' grounded extension cord is cut in half; the plug end supplies power to the outlets, and the socket end brings AC power back out, so you have someplace to plug in the transformer.

Algorithms and Encoding

The HC11 has a sophisticated timer section that is perfect for this application. The timer section has a 16-bit timer that counts at 2 MHz, and a set of four output compare registers that correspond to four output pins. To make an output go high at a point in the future, merely take the current timer count, add in the delay offset, and store it in the appropriate output compare register. The HC11 will then set that output high when the counts match, without any program intervention. This simplifies the code immensely; the code merely has to set up the appropriate offsets for all four channels, store them to the output compare registers, and then have the rest of the half-cycle for housekeeping.

At the default count rate, the timer overflows every 32 (ish) mS, but a half-cycle is only about 8 mS, so there's no chance of overflow in this application

My current implementation supports 64 dim levels. These dim levels are stored in a table which encodes the offset needed for each dim level. Each count is 0.5µS, so to dim halfway, the count for 4.16mS (half of a half cycle, or 1/240th of a second) would be 8,333.

The Main Loop

Wait until the zero-crossing signal is received. This is done by polling the input that the zero-crossing signal is connected to. Store the current timer count.

Force all the outputs to low. This insures that when we get to the zero-crossing point, the relays will turn off.

Take the stored time count and add the offset that will get us to the true zero-cross.

For each channel, add in the offset for the current dim level, and store it to the proper output compare channel

Figure out the next offset for each channel

Go to step 1.

Because we want to be able to have an offset of zero (no wait to turn-on), this code has to finish executing before we hit the true zero-cross. The current implementation is fast enough to do this, but if it wasn't, I'd simply skip dim level 0 (zero offset), and only let dim level 1 be the brightest one. With 64 dim levels, the difference isn't noticeable.

The HC11 handles everything for us once we've finished step 4, so step 5 has until the next zero-crossing signal to get set up for the next half-cycle. This is something on the order of 16,000 clocks, which is a lot of code.

If you were doing this with a microcontroller without output compare – or you wanted to do more than four channels with an HC11 – it would become more complicated. You could use the first period to generate the information for the next cycle (assuming you can get it done in 1,300 clocks; the width of the first period at 64 levels). If that wasn't enough, you could do it piecemeal (yuck), or, if your microcontroller supports timer interrupts, set up a timer interrupt for the first period, and then have the interrupt service routine turn on any channels that needed to be turned on, and set up the next interrupt. This would allow the code for the next channel to run during the non-interrupt times, but would make the code quite a bit more complex.

Running at the same time are some timekeeping functions that handle starting the animation when it gets dark (about 4:30 in the Seattle area), running 4.5 hours, and then turning off until the next day.

All the code is written directly in HC11 assembler. There is an SBASIC compiler available, which you might want to use. I found I could write the assembly code fairly quickly once I got into the HC11 mindset.

References

Botboard	http://www.rdrop.com/users/marvin/botboard/botboard.htm
68HC11	http://www.nwlink.com/~kevinro/products.html
HC11 Reference Manual	**(68HC11RM/AD)** http://www.motorola.com
Solid-State Relays	http://www.hut.fi/Misc/Electronics/circuits/semiconductor_relays.html

Encoding

One of the real challenges of this project is coming up with a minimal encoding for the animations. For this project, each step is encoded in seven bytes:

Byte	Description
1	Type of animation (dim or chaser)
2	# of loops for this step
3	Cycles to wait between loops
4-7	Channel information

For a dim animation, a typical encoding would be:

```
1   3F   14   01   00   FF   00
```

This means we should do this step for 3F loops, and that each loop should happen after 14 cycles (1/6th of a second). At each loop, we should add 1 to the channel 1 dim level, and add FF to channel 3, which is the same as subtracting 1 from it. So, this encoding will ramp channel 1 from its current dim level (which had better be 0, or we have problems) up to 3F, and channel 3 from 3F down to zero. This will take 3F * 1/6th = 10.5 seconds.

Generating Tables and Encodings

Generating the offset table for the dim levels and the animation encodings isn't something you'd want to do by hand. I've therefore written some Perl scripts that generate both the dimming table and the animation encodings, which are then combined with the code and assembled using asm11.

After the code is assembled, it is downloaded to the HC11 with a utility called DL11. The interface needed to connect the HC11 to a standard serial port is detailed in the BotBoard documentation.

Construction

The BotBoard is built following the instructions. I usually populate the board fully even though I

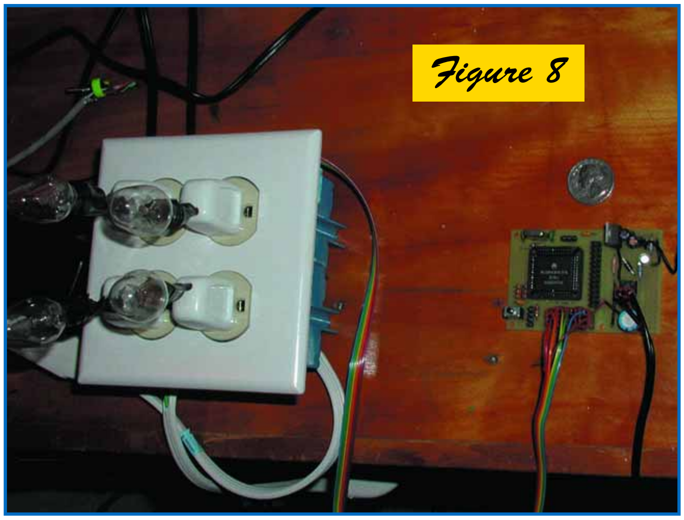


Figure 8

might not be using all the output pins at this time; it's easier to do now than later, and the extra headers are cheap.

In the prototype area next to the BotBoard, the power supply and zero-crossing circuits are built. I didn't breadboard the circuit first, which shows from my layout. It's not pretty, but luckily layout isn't critical for these circuits. The transformer connects to a three-pin header so it can be easily unplugged. See Figure 6.

The AC box holds the duplex outlets, with the bonding tab on the hot side broken off. The neutral wire for both outlets is hooked up, as are the grounds. A ribbon cable is hooked to the solid state relays, and then the whole business is placed in the box.

The appropriate connectors are then added to the ribbon cable. The BotBoard is designed to drive servos through these outputs, so the header locations aren't terribly convenient for this application. This required me to attach four individual three-pin connectors to the ribbon cable, and then use hot glue to create a single connector. See Figure 7. Figure 8 shows a picture of the whole project. I have night-light bulbs plugged into the outlets for debugging.

Debugging

Debugging an HC11 is interesting. I built a small status indicator out of a spare LED bar graph display I had lying around, and hooked it to a couple of four-pin headers. This can easily be slipped over the pins for the B port, so that debugging information can be written there. It's sometimes challenging to do debugging this way, but that's part of the fun.

It's also useful to generate your own signals; I used this to determine closely when the zero-cross pulse starts. A simple loop finds the pulse, and then it's easy to wait for a given number of clocks, and then turn on the B port, and turn it off a short time after. With this signal on the scope along with the DC signal, it was easy to determine the interval to within a few clocks (a couple of microseconds).

Conclusion

Once you have the project built, you'll need to write the controller code or use mine (available on the *Nuts & Volts* website). Then you'll have to deal with the lights, which usually takes me more time than the controller.

Going Further

I've had a few ideas on where to go from here. A four-channel X-10 dimmer seems fairly straightforward, and if you can do that, you could add X-10 relay control easily. I'm also interested in using the A/D capability of the HC11 to do something that responds to people or cars. Perhaps a Santa who turns his head to follow you when you go by … **NV**

Preprinted from December 2000 Nuts & Volts Magazine. All rights reserved. No duplication permitted without permission from T & L Publications, Inc.