Do It Up With DMX

By Jon Williams

For Nuts & Volts Magazine, Column 3, November 2009

Even if you've never heard of DMX512-A (DMX) chances are you've seen it in action. Where? At any large stage production, really: concerts and plays are big users of DMX-controlled lighting. So what if you're not one for concerts or the theater? Well, have you ever been to a night club with lots of crazy, dancing, pulsating lights? Then you've seen the magic of DMX. Not being one for dancing, if I do find myself in a club it's usually looking up, imagining the stream of data flying through the wire harnesses above to create all the lighting and movement that most others are oblivious to. Of course, I am sometimes startled when one of my friends yells, "Hey, Jon, what the heck are you looking at?" over the music!

So what is DMX? It is in fact a very simple, half-duplex (one direction, controller to fixture) serial protocol that runs over a standard RS-485 hardware link. The protocol was originally designed for controlling stage lighting, but as it is so easy to implement it has been put to use in a variety of show control applications.

We can break down the protocol into four essential elements:

- Break (B)
- Mark After Break (M)
- Start byte (S)
- Packet of Frames (Fx)

Figure 1 visualizes these elements as seen on the DMX RX pin of the Propeller.

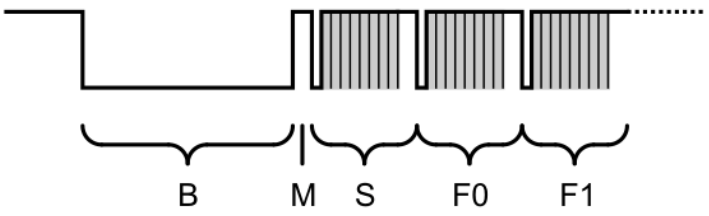

Figure 1

The *Break* is what allows all receivers on the system to synchronize themselves with the packet; this is a space (0) on the line that lasts 88us or longer. The *Mark After Break* (MAB) is a short rest with the line at idle (1); the MAB is 8 µs or longer. The first byte that follows the MAB is called the *Start* byte; it is typically zero and ignored in many systems (though it really shouldn't be). DMX bytes are transmitted in

8N2 (eight data bits, no parity, two stop bits) format. After the *Start* byte is the *Packet* of channel values, called *Frames* (0 to 255, also 8N2), which could be up to 512 bytes.

Light fades and motion are created by a master controller that streams the DMX data at a pretty swift clip: 250K baud. At this rate the *Break*, *MAB*, *Start* byte, and 512 *Frames* can be transmitted every 22.7 milliseconds (per the DMX specification).

I've written DMX receiver code for the SX28 but it is a challenge, especially when one needs to do brightness control of LEDs as we intend to do here – there's not a lot of room left in the interrupt when running an RX UART VP at 250K. With the Propeller and a dedicated DMX receiver cog, however, it's really very simple – so much so that it makes me shake my head and smile.

What we're going to build this month is a generic DMX IO add-on for the Propeller Platform with three channels of medium current output to control devices like 12 V LED circuits and small DC lamps. This will let us create a simple DMX lighting fixture using a high-brightness RGB LED.

DMX Hardware

Figure 2 shows the schematic for a generic DMX interface – yes, this circuit can transmit as well as receive. It would have been silly to design an add-on module for the Propeller Platform that couldn't transmit as well, especially since the "cost" of this upgrade was a resistor and a single IO pin. Pretty cheap price for the flexibility, don't you agree?

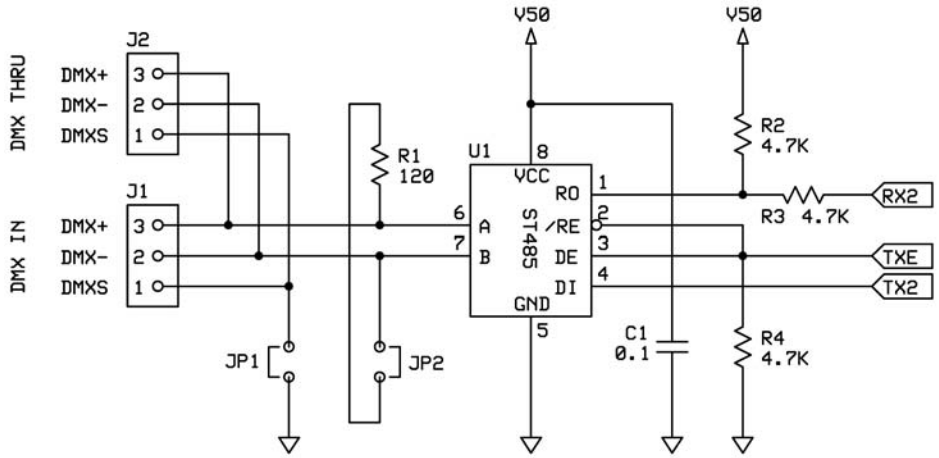

Figure 2

A quick note about JP1 and JP2: JP1 is used only when the node is the master (transmitting) and only one node will ever use JP1 (for receiver devices we leave this out). JP2 is for the end nodes (transmit or receive) on a DMX network; this resistor prevents reflections. So, if your DMX project using this circuit is the last on the DMX line then JP2 needs to be installed. For a lot of really great information on RS-485 hardware please see Jan Axelson's book, *Serial Port Complete*.

For what it's worth, my design does in fact violate the DMX specification in that I'm using 3-pin XLR connectors instead of the 5-pin units that are normally called for. But guess what? I have a mini DMX console and a popular DMX lighting fixture here in my office and both use 3-pin XLR connectors; this "violation" is pretty commonplace.

The circuit is a standard, half-duplex RS-485 interface that defaults to RX mode by pulling the /RE and DE lines low through resistor R4. You may be wondering why I went with a 5 V device when a 3.3 V device is available. Cost. There is a 5 V supply on the Propeller Platform and the cost of the 5 V

ST485BN plus resistor R3 is about half of the 3.3 V device. R3 limits the current into the RX2 pin (DMX RX input) of the Propeller. R2 holds the line high (idle state) when the ST485BN is set to transmit mode and the RO output goes Hi-Z (this resistor is required for projects that will do bi-directional comms). Finally, we don't have to worry about direct control of the TXE line with the Propeller as the minimum VIH level of the ST485BN is 2.0 volts.

Okay, then, let's have a look at the code. The heart of the object will of course run in its own cog, happily receiving DMX data on the assigned pin and writing it to an array that we can read from out top-level application. I've also added an activity output LED; this lights when receiving a frame byte so we know the line is active.

From the top, here's the setup and code that monitors the DMX RX line for the Break period.

```
dmxin           andn    outa, ledmask
                mov     dira, ledmask

                mov     ctra, NEG_DETECT
                add     ctra, rxpin
                mov     frqa, #1

waitbreak       waitpeq rxmask, rxmask
                mov     phsa, #0
                waitpne rxmask, rxmask

shortpacket     waitpeq rxmask, rxmask
                cmp     BREAK, phsa     wc
        if_ae   jmp     #waitbreak
```

On entry we make the LED pin an output and off and then setup **ctra** to count (at the system clock rate) whenever the DMX RX pin goes low (negative detect mode). This is an easy way to use the counter for pulse width timing; we simply check or reset the counter whenever the RX pin is high.

At waitbreak we begin by waiting for the RX line to go high using **waitpeq**. When it does we reset the **ctra** accumulator (**phsa**) and then wait for the RX line to go low. After it goes high again (so we've seen a high, low, then high) we can compare (**cmp**) the value in the counter's accumulator with the minimum timing for a break. If the pulse was short, i.e., not a valid break, the program will loop back to waitbreak and try again. Why would we have a short Break? We wouldn't, but we could power on in the middle of the Packet and we don't want to move unsynchronized values into our array. By waiting for the next Break we can get into sync with the DMX stream.

After we've detected a valid break we setup to receive up to 513 serial bytes. The first is the Start byte and will usually be zero. Still, we shouldn't ignore this byte; we should make it available to the application to check.

For review, a serial byte will have a start bit, eight data bits, and one or more stop bits; in DMX512-A each byte has two stop bits. Figure 3 shows the signal going into the DMX RX pin of the Propeller: the idle state of the line is high, a start bit is low (0), the data bits arrive LSB first and are read directly from the line, and the stop bits are at the line's idle state (1). The value in the diagram is $CF.

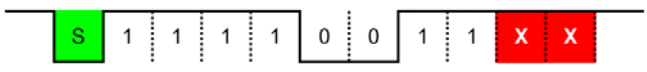

Figure 3

Another task to deal with is handling a short packet, that is, less than 512 frame bytes. A typical DMX controller will transmit the Start byte plus 512 frames, but it doesn't have to. For example, I have a mini,

6-channel controller that sends the Start byte plus six frames, and at a very low rate (every 100 ms). What I'm getting at is we'll have to smarten-up our serial receive code to detect a new break, even when we don't expect one.

```
getpacket       mov     bufpntr, buf0
                mov     count, PACKET

rxbyte          mov     phsa, #0
                mov     rxwork, #0
                mov     rxcount, #8
                mov     rxtimer, US_006

                waitpne rxmask, rxmask
                add     rxtimer, cnt
                or      outa, ledmask

rxbit           waitcnt rxtimer, US_004
                test    rxmask, ina     wc
        if_c    mov     phsa, #0
                shr     rxwork, #1
                muxc    rxwork, #%1000_0000
                djnz    rxcount, #rxbit

breakcheck      waitcnt rxtimer, #0
                test    rxmask, ina     wz
                andn    outa, ledmask

        if_z    jmp     #shortpacket

                wrbyte  rxwork, bufpntr
                add     bufpntr, #1
                djnz    count, #rxbyte

                jmp     #waitbreak
```

At the top we set *bufpntr* to the hub address of the array that will hold the DMX data, and *count* to the number of bytes to receive (513). At rxbyte the Break timer (**ctra**) is reset and we prep for receiving a serial byte that has a bit timing of 4 µs (250K baud). The bit timer is initially set to 6 µs (1.5 bits) so that we can position the bit sampling in the middle of the first bit; this timer will be activated (via syncing with **cnt**) on detection of the start bit. Note, too, that when the start bit is detected the LED is turned on to indicate DMX activity.

The bulk of serial receive code is at rxbit. After the bit timer expires the DMX RX line is sampled with **test** and the RX bit value is moved into the C flag. If the C flag is set (1) then we can restart the Break timer – remember this is setup to auto-increment whenever the DMX RX line is low. After the Break timer is dealt with the output value in *rxwork* is shifted right by one bit and we move the new bit from C into bit7 using **muxc**. If there are more bits to receive then the code loops back to rxbit, otherwise it will drop through to breakcheck.

The program waits one more bit period (4 µs) and then samples the line again, saving the result into the Z flag. At this point we should be sampling a stop bit which is high. If, however, we have a new Break period then the line will be low (0) and the Z flag is set. When this happens the program jumps back to shortpacket which completes and validates the timing of the new Break period.

Most of the time, though, we have a valid byte which is moved into the DMX array using **wrbyte**. Once the entire DMX packet has been received the program jumps back to waitbreak and starts all over.

So there you have it, a DMX receiver engine in under 40 [working] lines of code. Of course, there is an interface in Spin which lets us set the RX and LED pins, and takes care of setting all the timing

parameters to match the system clock. After the init() method we'll use the read() method to grab a byte from the DMX stream. Remember, byte 0 will be the DMX Start byte, bytes 1 through 512 will be the DMX frames.

Before moving on, let me point out that there is a DMX object in the Propeller Object Exchange by Tim Sweeter (www.brilldea.com) that is very advanced, providing all kinds of interesting statistics about the DMX stream (e.g., the length of the break, length of MAB, actual number of frames transmitted, etc.). If you're feeling brave please have a look – you're sure to learn some new Propeller tricks. Tim's code will of course work on the P/P DMX IO board, and is very useful for creating a DMX diagnostics device.

LED Modulation without Legal Hassles

Believe it or not, there is a patent for LED modulation via PWM. No kidding. Tragically, it's patents on wholly-unoriginal ideas like this that throw the credibility of the patent system into question (*he laments as he looks up at the three patent plaques on the office wall*). In my opinion, the examiner of that particular patent should have his/her head examined. A patent for controlling the brightness of an LED via PWM. Puhleeze.... What's next, patenting Ohm's Law? Yes, I'm being a facetious but Google was actually granted a patent for their search engine page layout!

Of course, the LED PWM patent owner has a completely different point-of-view and it is in their financial interest to aggressively protect the patent from interlopers and doing what they can to collect licensing fees. Don't believe it's true? I got an email from a Propeller forum member telling me that his company had received a "cease and desist" letter from said patent owner. He then went on to mention a public-domain technique called Bit Angle Modulation (BAM) which, at that time, I had never heard of.

Now, I'm not a lawyer, and I've not yet played one on TV or in a movie, so take all of this stuff with a grain of salt. That said, my forum friend has no reason whatsoever to lead me astray; he was kind enough to review my circuit and pointed out that I could dump a couple spare resistors that really weren't really doing anything. At the very least I thought I should research BAM as I have a possible client project that involves brightness control of high-power LEDs and I would hate for him to receive one of *those* letters.

A few minutes of web research turned up a whole lot of hoopla surrounding BAM, so much so that I nearly dismissed it all as hype. Once I got beyond my temporary bad attitude and started coding it all made sense, and writing a three-channel BAM driver for this month's project became a pretty simple exercise.

So what is BAM? It's a modulation strategy that uses a bit's position within a value as the basis for the timing of that position. For example, bit 0 is output and then we wait one 'period.' Next we output bit 1 and hold for two periods. After that we output bit 2 and wait four periods. You should start to see the pattern: the timing for a given bit is $2^{bit\#}$ periods. Following this pattern the final bit in a byte, bit 7, has a timing value of 128 periods. In the end, the complete timing for one byte will be 255 periods, but we only had to deal with eight cycles (one per bit). The fact that we only have to deal with eight cycles (versus 256 with traditional PWM) is source of most of the hoopla surrounding BAM.

Figure 4 shows the weighted timing in a 4-bit BAM value. As you can see, bit 3 gets eight timing periods, bit 2 gets four periods, etcetera. In Figure 5 you can see what the output looks like when the BAM value is set to 10. I think it's the asymmetrical output (for most values) that helps BAM get around the LED PWM patent.

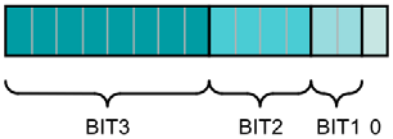

Figure 4

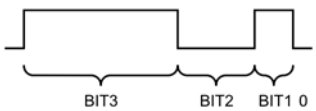

Figure 5

Most of the code you'll find on the Internet shows how to setup a variably-timed interrupt to handle the bit timing. The Propeller doesn't have or use interrupts so we're going to take another route – one that turns out to be deceptively easy. Here's my implementation of BAM for the Propeller.

```
bamrgb          or      dira, rmask
                or      dira, gmask
                or      dira, bmask

                min     TIX_001, #121

bamstart        mov     bitmask, #%0000_0001
                mov     bitperiod, TIX_001
                mov     bittimer, bitperiod
                add     bittimer, cnt

                mov     tmp1, #0

getlevels       rdbyte  rlevel, rpntr
                rdbyte  glevel, gpntr
                rdbyte  blevel, bpntr

bamloop         test    rlevel, bitmask wc
                muxc    tmp1, rmask
                test    glevel, bitmask wc
                muxc    tmp1, gmask
                test    blevel, bitmask wc
                muxc    tmp1, bmask
                mov     outa, tmp1

                shl     bitmask, #1
                and     bitmask, #$FF   wz
        if_nz   shl     bitperiod, #1
        if_z    mov     bitmask, #%0000_0001
        if_z    mov     bitperiod, TIX_001

                waitcnt bittimer, bitperiod

        if_nz   jmp     #bamloop
                jmp     #getlevels
```

On entry we make the RGB pins outputs and do a quick check to ensure that the timing period is long enough to get the work done. You'll recall that using **waitcnt** is easy and convenient, but if we are short of the value and get a rollover we end up waiting nearly a minute – not what we want to have happen here. The **min** instruction takes care of fixing a value that the user may have mismanaged in the interface section. I found the value of 121 estimating (counting cycles) and then empirical testing until it "broke."

At bamstart we create a mask for bit 0, set the timing for bit 0 (one "period"), start a timer, and initialize a value that will hold the new outputs for each cycle.

The real work begins at getlevels where we read the RGB values from the hub. At bamloop we test each color value against *bitmask*, writing the current bit value to the C flag. Using **muxc** and the channel mask for the color, the bit value is written to *tmp1* which is finally moved to the outputs. By using *tmp1* all outputs are simultaneously updated.

The bit-under-test mask is then shifted left and **and**ed with $FF – if the result of the **and** operation is zero then we're done with the loop; all eight bits have been processed which means we can reset the test mask and the bit period. Until this happens we shift the value of *bitperiod* left by one which doubles its value; this is exactly what we need for the next higher bit.

After the timer expires it is reset with *bittimer* and we jump back to bamloop for the next bit or to getlevels if it's time to start a new cycle. Done! Of course you can add channels if you like, just be sure to update the minimum timing for a bit period so that you can get all the work done within the cycle.

A Simple DMX RGB Lighting Fixture

With the hard work (which really wasn't very hard) out of the way we can create a simple, 3-channel DMX lighting fixture. For this we'll need a device address switch which corresponds to the first of our three channels – Figure 6 shows a 9-position DIP switch wired to P0..P8 of the Propeller. No need to worry about the reverse bit order (which was done for PCB layout convenience); Spin has a neat trick which will take care of this for us.

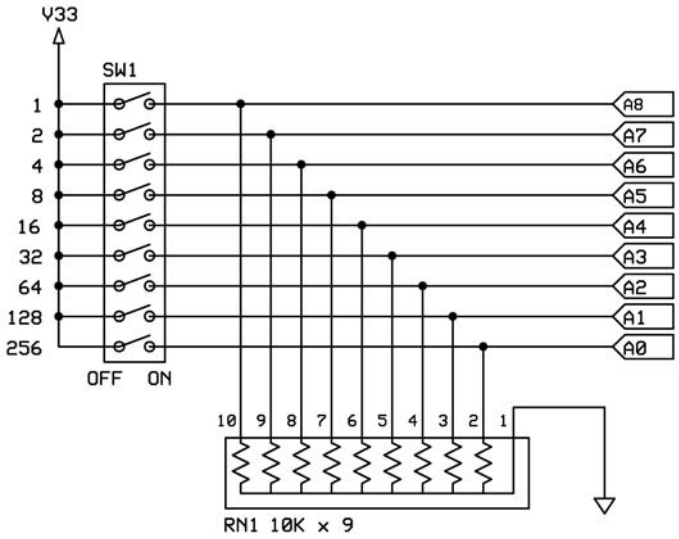

Figure 6

For my first project I wanted to drive a 1W RGB LED that a friend gave me, but I also wanted to run small DC lamps for other projects. In order to accommodate both the circuit uses a moderate-current, high-side driver (Figure 7) on each output. Some of you will quickly point out that the final output transistor is missing a pull-up to Vin. Not so. This transistor is actually a TIP125 Darlington and, as you can see in Figure 8, the base has an internal 8.12K path to the emitter – we're covered, and with the TIP125 we can control plenty of current.

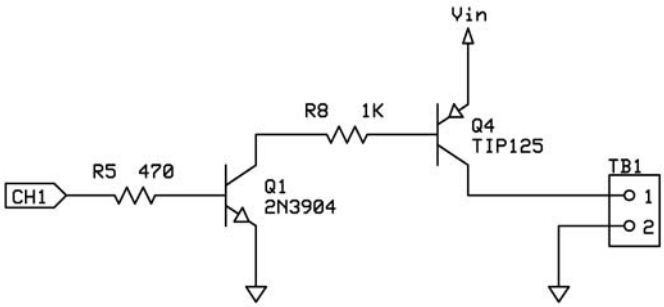

Figure 7

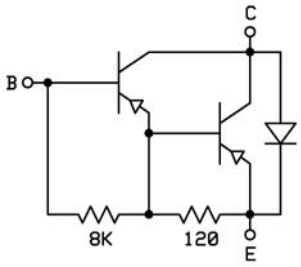

Figure 8

Okay, here's the code for the 3-channel DMX fixture.

```
pub main | dmxstart, chan, level

  dmx.init(24, 16)
  rgb.init(9, 10, 11)

  repeat
    chan := ina[0..8]
    if (chan > 0) & (dmx.read(0) == 0)
      level := dmx.read(chan++)
      rgb.setred(rgb.ezlog(level))
      level := dmx.read(chan++)
      rgb.setgreen(rgb.ezlog(level))
      level := dmx.read(chan)
      rgb.setblue(rgb.ezlog(level))
    else
      rgb.setall(0)
```

I'm not kidding, that's all we need to knit the DMX receiver and BAM output objects together to create a DMX-compatible lighting fixture.

We start by initializing the DMX and BAM objects and then drop into a **repeat** loop. The first line in the loop is really cool. What this does is reads bits 0 through 8 of the inputs, masking off the others, and flipping their order (bit 0 to bit 8, bit 8 to bit 0) – in one fell swoop we've read the switch bits and put them into the correct order, giving us the current channel setting. Come on, you've got to think that's cool! Just so we're clear, we know that the bit order is reversed because the LSB (0) is in the MSB position within the brackets. Another bonus that the switch could have used any set of IO pins; so long as the group was contiguous. The only pins read are those defined within the brackets, and the result bits are flipped and shifted (to zero-align) if required. We can use this technique to read any parallel inputs and get a value that is immediately usable.

If the channel switch is valid (not zero) and the DMX Start byte is valid (zero) then we read the channel values from the DMX array and move them to the outputs using BAM to handle the brightness modulation. You can see that Spin borrows heavily from C in that we can use the post-increment operator (++) on *chan*; this lets us read the current channel and then update that variable before moving on.

Visually, LEDs can seem a little harsh when using straight linear values as we'd get from the DMX stream. A friend showed me a cute trick that creates a logarithmic curve: you simply square the value and then divide it by 256. Doing this makes the LED brightness output much more appealing. The math for this trick is wrapped up in a method called .ezlog() that is part of the BAM object. For you advanced users there is also a method to read a value from a table which allows you to map the DMX input value to an output value as desired.

If Three is Good, Four Must Be Better!

My original design (see the prototype in Figure 9) was oriented toward RGB lighting control and as I was building it a post in the Propeller forum vis-à-vis stepper motors got me thinking: *If I added one more channel I could control a unipolar stepper motor with the TIP125 outputs*. Then I thought: *Why not add servo headers on the outputs as well?* So, for those of you who purchase the PCB or kit through Gadget Gangster (recommended) you'll get the four-channel version which gives you more options for outputs. Of course, if you want to roll you own I've included the 4-channel files in the PCB download so you can use them as you please.

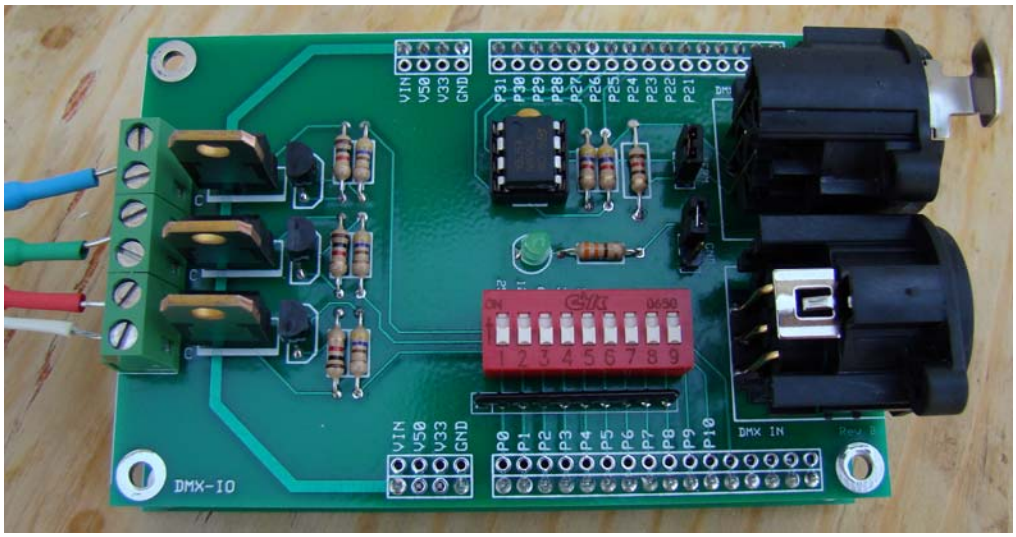

Figure 9

Okay, then, how about adding a little DMX to your holiday lighting arsenal? It could be a lot of fun and really make the neighbors jealous! Until next time – and next year! – here's to *spinning* and *winning* with the Propeller.

Bill of Materials

C1 0.1uF	Mouser 80-C315C104M5U	
J1	XLR-3M	Mouser 523-AC3MAH-AU-B
J2	XLR-3F	Mouser 523-AC3FAH-AU-B
JP1-JP2	0.1 M-STRT	Mouser 517-6111TG
Jumpers		Mouser 517-950-00
LED1	3mm Green	Mouser 859-LTL-4231
Q1-Q3	2N3904	Mouser 863-2N3904G
Q4-Q6	TIP125	Mouser 512-TIP125
R1	120	Mouser 293-120-RC
R2-R4	4.7K	Mouser 291-4.7K-RC
R5-R7	470	Mouser 291-470-RC
R8-R10	1K	Mouser 291-1K-RC
R11	330	Mouser 291-330-RC
RN1	10K x 9	Mouser 71-CSC10A01-10K
SW1	DIP x 9	Mouser 611-BD09
TB1-TB3	5mm term block	Mouser 571-2828362
U1	ST485BN	Mouser 511-ST485BN
X1-X4	0.1 M-STRT	Mouser 517-6111TG
PCB		GadgetGangster.com ExpressPCB.com
Parts Kit		GadgetGangster.com