Wireless Game Show Poppers for the Classroom!

Created by itaparcade

https://learn.adafruit.com/wireless-game-show-poppers

Last updated on 2023-08-29 02:38:16 PM EDT

Table of Contents

Overview

Is it time for school already? Need ways to keep students engaged but on a tight budget? We have a $15.00 game show system for you based on a combination of wireless IR poppers and game show software for PC and Mac computers. All you need are 1) the Duo Pop for iPad Game () , 2) a 5 volt Pro Trinket Arduino-compatible microcontroller (), 3) Game Show Presenter Software () (free demo version available) to create your questions as well as host your game, and 4) a little bit of time to hack and solder on the Duo Pop for iPad Receiver. Once finished with this project, you will have a popper USB receiver system that is completely hackable allowing you to interface the poppers with all kinds of other learning software for both PC and Mac computers. Are you ready to hack the Duo Pop? Before hacking, let's discuss the parts that you will need for this project.

Use a Pro Trinket 5 Volt 16 MHz for this Project since the Virtual USB for Arduino Library is based on the UNO that operates at 16 MHz

Gathering Parts

You will need several items to get started and will eventually need a soldering iron. The parts list below are used for completing this project.

1. Duo Pop for iPad Game ()
2. Pro Trinket 5 Volt ()
3. Game Show Presenter Software ()

Based on this list, you can build this wireless popper system with a USB receiver using a demo version of the game show software or other software that might be available to you. If interested in a preassembled kit or a preprogrammed Pro Trinket, send us an e-mail at info@itaparcade.com.

We like to divide our projects into sections based on 1) Learning, 2) Building, 3) Programming, and 4) Playing. This allows you to pick and choose which sections you want to focus on based on your previous making experiences. Let's check out the parts list for each section.

Learning Section:

1. Arduino Uno (http://adafru.it/50)
2. IR Sensor (http://adafru.it/157)
3. Duo Pop Poppers

Building Section:

1. Duo Pop IR Receiver
2. Pro Trinket (http://adafru.it/2000)
3. Diagonal Cutters (http://adafru.it/152)
4. Soldering Iron (http://adafru.it/180)
5. Drill and 1/16 drill bit
6. #2 x 3/8 inch Mounting Screw

Programming Section:

1. Setting up Arduino IDE ()
2. V-USB for Arduino Library (version 5) ()
3. USB cable (http://adafru.it/592)

Playing Section:

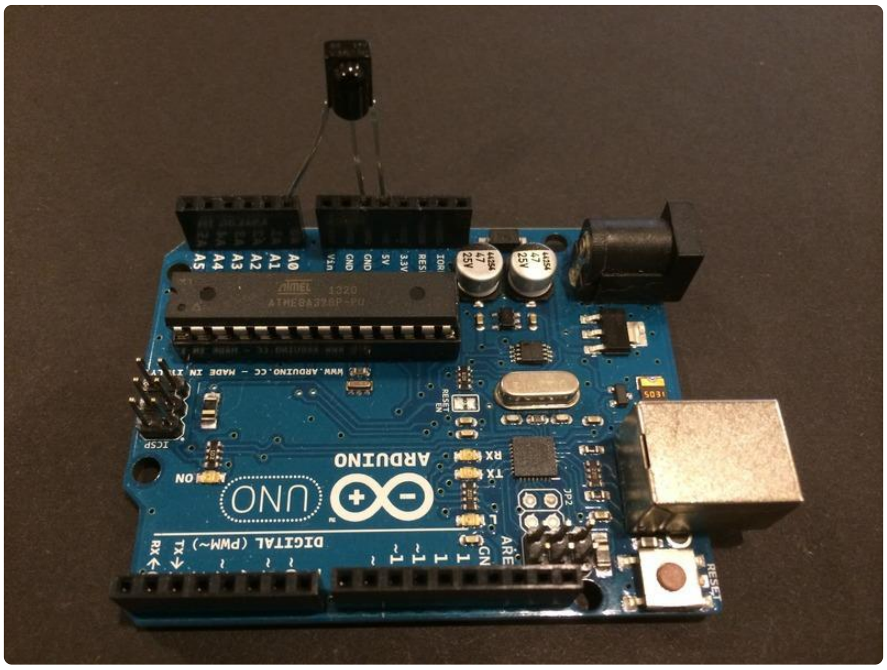
1. Game Show Presenter Software ()
2. Completed Duo Pop USB Receiver
3. USB cable (http://adafru.it/592)

Learn

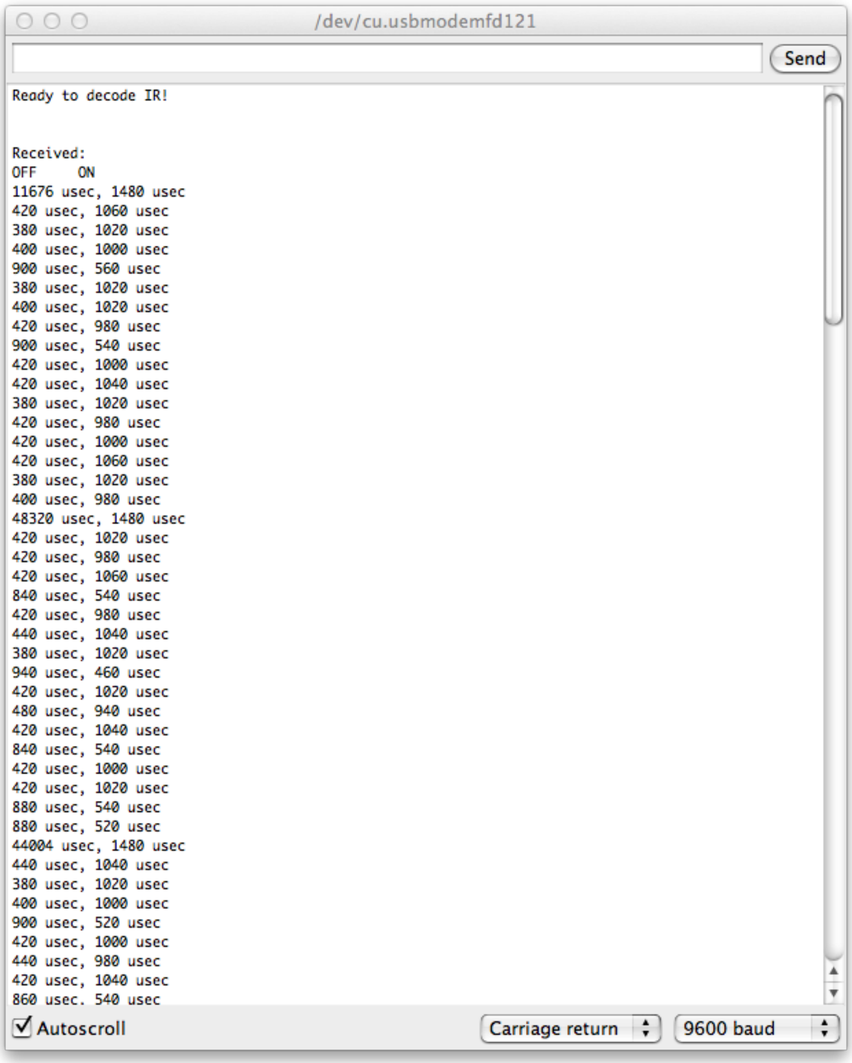
Decoding IR Popper Transmissions

Since the Duo Pop Poppers are based on IR technology to transmit player responses, there is a good chance that the transmission frequency is around 38 kHz. To test this hypothesis, we first followed the Using an IR Sensor () tutorial that leveraged an IR sensor and an Arduino UNO. We did however make a slight modification to the code so we could eliminate having to need hook up wires between the IR sensor and the Arduino UNO. The picture below illustrates our setup using an IR sensor and the Arduino UNO. Notice the OUT pin of the IR sensor is now connected to A0 compared to the original connection diagram. Based on this change, change the following two lines of code in the Using an IR Sensor () tuorial to test the detection of the Popper IR transmissions with the Arduino UNO.

```
// Comment out #define IRpin_PIN PIND
// Comment out #define IRpin 2

// Add these two lines instead
#define IRpin_PIN PINC
#define IRpin     PINC0
```



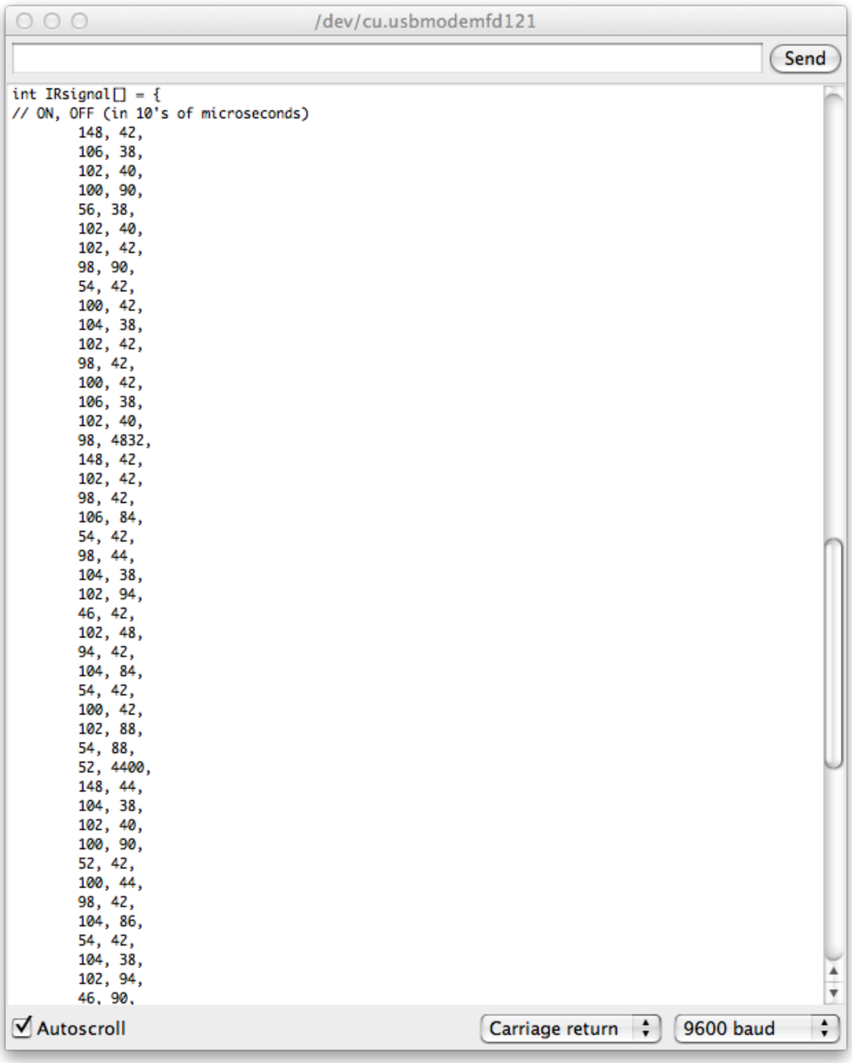
After loading the sketch from the Using and IR Sensor tutorial and making the two line changes suggested above, open the Serial Monitor from your Arduino IDE and press the blue Duo Pop Popper. The first part of your screen should look like below.



```
Ready to decode IR!


Received:
OFF      ON
11676 usec, 1480 usec
420 usec, 1060 usec
380 usec, 1020 usec
400 usec, 1000 usec
900 usec, 560 usec
380 usec, 1020 usec
400 usec, 1020 usec
420 usec, 980 usec
900 usec, 540 usec
420 usec, 1000 usec
420 usec, 1040 usec
380 usec, 1020 usec
420 usec, 980 usec
420 usec, 1000 usec
420 usec, 1060 usec
380 usec, 1020 usec
400 usec, 980 usec
48320 usec, 1480 usec
420 usec, 1020 usec
420 usec, 980 usec
420 usec, 1060 usec
840 usec, 540 usec
420 usec, 980 usec
440 usec, 1040 usec
380 usec, 1020 usec
940 usec, 460 usec
420 usec, 1020 usec
480 usec, 940 usec
420 usec, 1040 usec
840 usec, 540 usec
420 usec, 1000 usec
420 usec, 1020 usec
880 usec, 540 usec
880 usec, 520 usec
44004 usec, 1480 usec
440 usec, 1040 usec
380 usec, 1020 usec
400 usec, 1000 usec
900 usec, 520 usec
420 usec, 1000 usec
440 usec, 980 usec
420 usec, 1040 usec
860 usec, 540 usec
```

If you scroll down until you see the int IRSignal[] portion of the output, you should see something similar to below. Repeating this process for all four of the poppers allowed us to figure out the transmissions coming from the poppers. How would you now create a decoder based on this information? It is discussed below.

The values may not be exactly the same due to variances in measurements associated with the transmission pulses



```
int IRsignal[] = {
// ON, OFF (in 10's of microseconds)
        148, 42,
        106, 38,
        102, 40,
        100, 90,
        56, 38,
        102, 40,
        102, 42,
        98, 90,
        54, 42,
        100, 42,
        104, 38,
        102, 42,
        98, 42,
        100, 42,
        106, 38,
        102, 40,
        98, 4832,
        148, 42,
        102, 42,
        98, 42,
        106, 84,
        54, 42,
        98, 44,
        104, 38,
        102, 94,
        46, 42,
        102, 48,
        94, 42,
        104, 84,
        54, 42,
        100, 42,
        102, 88,
        54, 88,
        52, 4400,
        148, 44,
        104, 38,
        102, 40,
        100, 90,
        52, 42,
        100, 44,
        98, 42,
        104, 86,
        54, 42,
        104, 38,
        102, 94,
        46, 90,
```

The IR decoding tutorial was important because it indeed validated the transmission frequency of 38 kHz of the the Popper transmission signals while allowing us to decode the unknown transmission sequences from the poppers. The IR decoding tutorial enabled us to easily reverse engineer the IR transmissions from each of the poppers so we could code our own custom decoder. This naturally leads into following the second tutorial associated with Reading IR commands (). It is left as an exercise for you to modify the Reading IR commands () tutorial so that you can distinguish between the different color poppers. Have fun!

OK.....I couldn't just leave you hanging like that. Below is the Arduino code we converged to after leveraging code and lessons learned from the Reading IR commands tutorial. After you look at the code, program the Arduino UNO and open the Serial Monitor from the Arduino IDE. Press the different colored poppers and ask yourself why did we decide to create the decoder code the way we did? HINT: Has something to do with error control coding. This is a great time to learn about how wireless communication systems add redundancy in their transmissions to make sure the transmitted message has a high probability of getting to where it needs to be.

```
/* Raw IR decoder sketch!

 This sketch/program uses the Arduno and a PNA4602 to
 decode IR received. This can be used to make a IR receiver
 (by looking for a particular code)
 or transmitter (by pulsing an IR LED at ~38KHz for the
 durations detected

 Code is public domain, check out www.ladyada.net and adafruit.com
 for more tutorials!

 Company: itaparcade.com
 Twitter: @iTapArcade
 Email: info@itaparcade.com
 Date: 10/1/2014
 Modified this sketch to decode the Duo Pop Poppers from Duo Pop for iPad Game
 with an Arduino UNO and IR receiver
*/


// We need to use the 'raw' pin reading methods
// because timing is very important here and the digitalRead()
// procedure is slower!
//uint8_t IRpin = 2;
// Digital pin #2 is the same as Pin D2 see
// http://arduino.cc/en/Hacking/PinMapping168 for the 'raw' pin mapping
#define IRpin_PIN      PINC
#define IRpin          PINC0
#define LED            1

// the maximum pulse we'll listen for - 65 milliseconds is a long time
//#define MAXPULSE 65000
#define MAXPULSE 1000

// what our timing resolution should be, larger is better
// as its more 'precise' - but too large and you wont get
// accurate timing
#define RESOLUTION 20

// What percent we will allow in variation to match the same code \\ was #define
FUZZINESS 20

#define FUZZINESS 45

// we will store up to 100 pulse pairs (this is -a lot-)
uint16_t pulses[100][2];  // pair is high and low pulse
uint8_t currentpulse = 0; // index for pulses we're storing

// Red Code

int DUOPOPsignalRed[] = { // ON, OFF (in 10's of microseconds)
140, 50,
100, 50,
100, 100,
50, 50,
100, 50,
100, 50,
100, 100,
50, 50,
100, 50,
100};

// Blue Code

int DUOPOPsignalBlue[] = { // ON, OFF (in 10's of microseconds)
140, 50,
100, 50,
100, 50,
100, 100,
50, 50,
100, 50,
100, 50,
100, 100,
50, 50,
100};

// Yellow Code

int DUOPOPsignalYellow[] = { // ON, OFF (in 10's of microseconds)
140,
100, 50,
50, 100,
100, 50,
50, 100,
100, 50,
50, 100,
100, 50,
50, 100,
50, 100};

// Green Code

int DUOPOPsignalGreen[] = { // ON, OFF (in 10's of microseconds)
140,
50, 100,
100, 50,
50, 100,
100, 50,
50, 100,
100, 50,
50, 100,
100, 50,
50, 100};


void setup(void) {

  Serial.begin(9600);
  Serial.println("Ready to decode IR!");
  pinMode(IRpin, INPUT);

}



void loop(void) {

  int numberpulses;

  numberpulses = listenForIR();

//  Serial.print("Heard ");
//  Serial.print(numberpulses);
//  Serial.println("-pulse long IR signal");
  if (numberpulses == 10)
  {
   // Serial.println("Received IR!");
   // digitalWrite(LED, HIGH);
   //   delay(50);
    //  digitalWrite(LED, LOW);
  if (IRcompare(numberpulses, DUOPOPsignalRed)) {
    Serial.println("RED Buzzed In");
  }
  if (IRcompare(numberpulses, DUOPOPsignalBlue)) {
    Serial.println("BLUE Buzzed In");
  }
    if (IRcompare(numberpulses, DUOPOPsignalYellow)) {
```

```
    Serial.println("Yellow Buzzed In");
  }
   if (IRcompare(numberpulses, DUOPOPsignalGreen)) {
    Serial.println("Green Buzzed In");
  }
  }
}

int listenForIR(void) {
  currentpulse = 0;

  while (1) {
    uint16_t highpulse, lowpulse;  // temporary storage timing
    highpulse = lowpulse = 0; // start out with no pulse length

//  while (digitalRead(IRpin)) { // this is too slow!
   // while (IRpin_PIN &amp; (1 &lt;&lt; IRpin)) {
     while (IRpin_PIN &amp; _BV(IRpin)) {
       // pin is still HIGH

       // count off another few microseconds
       highpulse++;
       delayMicroseconds(RESOLUTION);

       // If the pulse is too long, we 'timed out' - either nothing
       // was received or the code is finished, so print what
       // we've grabbed so far, and then reset
       if (((highpulse &gt;= MAXPULSE) &amp;&amp; (currentpulse != 0)) ||
(currentpulse == 10)) {
         return currentpulse;
       }
    }
    // we didn't time out so lets stash the reading
    pulses[currentpulse][0] = highpulse;

    // same as above
    while (! (IRpin_PIN &amp; _BV(IRpin))) {
       // pin is still LOW
       lowpulse++;
       delayMicroseconds(RESOLUTION);
       if (((lowpulse &gt;= MAXPULSE)  &amp;&amp; (currentpulse != 0)) ||
(currentpulse == 10)) {
         return currentpulse;
       }
    }
    pulses[currentpulse][1] = lowpulse;

    // we read one high-low pulse successfully, continue!
    currentpulse++;
  }
}


boolean IRcompare(int numpulses, int Signal[]) {

  for (int i=0; i&lt; numpulses-1; i++) {
    int oncode = pulses[i][1] * RESOLUTION / 10;
    int offcode = pulses[i+1][0] * RESOLUTION / 10;


 //   Serial.print(oncode); // the ON signal we heard
 //   Serial.print(" - ");
 //   Serial.print(Signal[i*2 + 0]); // the ON signal we want


    // check to make sure the error is less than FUZZINESS percent
    if ( abs(oncode - Signal[i*2 + 0]) &lt;= (Signal[i*2 + 0] * FUZZINESS / 100)) {
  //    Serial.print(" (ok)");
```

```
    } else {
  //    Serial.print(" (x)");
      // we didn't match perfectly, return a false match
      return false;
    }


 //   Serial.print("  \t"); // tab
 //   Serial.print(offcode); // the OFF signal we heard
 //   Serial.print(" - ");
 //   Serial.print(Signal[i*2 + 1]); // the OFF signal we want


    if ( abs(offcode - Signal[i*2 + 1]) &lt;= (Signal[i*2 + 1] * FUZZINESS / 100)) {
  //    Serial.print(" (ok)");
    } else {
  //    Serial.print(" (x)");
      // we didn't match perfectly, return a false match
      return false;
    }

   // Serial.println();
  }
  // Everything matched!
  return true;
}
```

Once we were able to figure out the wireless IR transmission signals coming from all four of the poppers to distinguish between Blue, Red, Yellow, and Green using the Arduino UNO, we leveraged this same Arduino code on the Pro Trinket 5 volt at 16 MHz since the Pro Trinket can be thought of as a miniaturized Arduino Uno (both use the same Atmega chip) without the serial-to-USB bridge. However, we don't have a dedicated USB chip to interface with the computer using a Pro Trinket. This requires us making a virtual one based on the V-USB technology.

Note: The most tedious part of this project was decoding the wireless IR signals from the Duo Pop Poppers using the IR receiver sensor and the Arduino UNO. The adafruit.com tutorials were essential to us completing this project quickly. Make sure to go through those tutorials mentioned in this project.

Exploring V-USB Technology

Once we were able to successfully decode and determine which popper was being detected based on their unique codes, we focused on determining how to translate this IR decoder into USB keyboard strokes. We needed this capability to make sure we could inform the gameshow software as to who pressed their popper first. We leveraged lessons learned from the Trinket USB keyboard tutorial () to understand how we could use the Pro Trinket USB interface pins (Pins #2 and #7) to emulate a virtual USB keyboard. A similar V-USB keyboard Arduino project was conducted several years ago based on an Arduino V-USB library () leveraging discrete components to connect to the USB data bus. Since the Pro Trinket already has this circuitry built-in, it was just a matter of using the correct Pro Trinket USB pins to emulate a USB keyboard via the V-USB protocol. We leveraged version 5 of this library and added the USBKeyboard folder to our Arduino IDE library environment () with some slight modifications to the usbconfig.h file. The changes are shown below so the correct Pro Trinket USB interface pins are properly defined (Pins #2 and #7).

You can also change the vendor id, name, and other USB keyboard parameters, but be sure to comply with the V-USB Vendor ID and Product ID software policies with respect to these changes.

```
// Comment out #define USB_CFG_DMINUS_BIT      4
/* This is the bit number in USB_CFG_IOPORT where the USB D- line is connected.
 * This may be any bit in the port.
 */
// Comment #define USB_CFG_DPLUS_BIT       2

// Replace with lines below in usbconfig.h to work with Pro Trinket
#define USB_CFG_DMINUS_BIT      7
/* This is the bit number in USB_CFG_IOPORT where the USB D- line is connected.
 * This may be any bit in the port.
 */
#define USB_CFG_DPLUS_BIT       2
```

For completeness, I have included the usbconfig.h code that I modified to create the USB keybaord. Make sure to replace the V-USB version with this code to get started for testing purposes. You can then make modifications as necessary based on your project specifics. Just make sure if you change the vendor id, name, and other USB keyboard paramters that you comply with the V-USB Vendor ID and Product ID software policies.

```
/* Name: usbconfig.h
 * Project: V-USB, virtual USB port for Atmel's(r) AVR(r) microcontrollers
 * Author: Christian Starkjohann
 * Creation Date: 2005-04-01
 * Tabsize: 4
 * Copyright: (c) 2005 by OBJECTIVE DEVELOPMENT Software GmbH
 * License: GNU GPL v2 (see License.txt), GNU GPL v3 or proprietary
(CommercialLicense.txt)
 * This Revision: $Id: usbconfig-prototype.h 767 2009-08-22 11:39:22Z cs $
 */

#ifndef __usbconfig_h_included__
#define __usbconfig_h_included__

/*
General Description:
This file is an example configuration (with inline documentation) for the USB
driver. It configures V-USB for USB D+ connected to Port D bit 2 (which is
also hardware interrupt 0 on many devices) and USB D- to Port D bit 4. You may
wire the lines to any other port, as long as D+ is also wired to INT0 (or any
other hardware interrupt, as long as it is the highest level interrupt, see
section at the end of this file).
+ To create your own usbconfig.h file, copy this file to your project's
+ firmware source directory) and rename it to "usbconfig.h".
+ Then edit it accordingly.
*/
```

```
/* ---------------------------- Hardware Config --------------------------- */

#define USB_CFG_IOPORTNAME      D
/* This is the port where the USB bus is connected. When you configure it to
 * "B", the registers PORTB, PINB and DDRB will be used.
 */
#define USB_CFG_DMINUS_BIT      7
/* This is the bit number in USB_CFG_IOPORT where the USB D- line is connected.
 * This may be any bit in the port.
 */
#define USB_CFG_DPLUS_BIT       2
/* This is the bit number in USB_CFG_IOPORT where the USB D+ line is connected.
 * This may be any bit in the port. Please note that D+ must also be connected
 * to interrupt pin INT0! [You can also use other interrupts, see section
 * "Optional MCU Description" below, or you can connect D- to the interrupt, as
 * it is required if you use the USB_COUNT_SOF feature. If you use D- for the
 * interrupt, the USB interrupt will also be triggered at Start-Of-Frame
 * markers every millisecond.]
 */
#define USB_CFG_CLOCK_KHZ       (F_CPU/1000)
/* Clock rate of the AVR in kHz. Legal values are 12000, 12800, 15000, 16000,
 * 16500 and 20000. The 12.8 MHz and 16.5 MHz versions of the code require no
 * crystal, they tolerate +/- 1% deviation from the nominal frequency. All
 * other rates require a precision of 2000 ppm and thus a crystal!
 * Default if not specified: 12 MHz
 */
#define USB_CFG_CHECK_CRC       0
/* Define this to 1 if you want that the driver checks integrity of incoming
 * data packets (CRC checks). CRC checks cost quite a bit of code size and are
 * currently only available for 18 MHz crystal clock. You must choose
 * USB_CFG_CLOCK_KHZ = 18000 if you enable this option.
 */

/* ----------------------- Optional Hardware Config ------------------------ */

/*#define USB_CFG_PULLUP_IOPORTNAME   D*/
/* If you connect the 1.5k pullup resistor from D- to a port pin instead of
 * V+, you can connect and disconnect the device from firmware by calling
 * the macros usbDeviceConnect() and usbDeviceDisconnect() (see usbdrv.h).
 * This constant defines the port on which the pullup resistor is connected.
 */
/*#define USB_CFG_PULLUP_BIT          5*/
/* This constant defines the bit number in USB_CFG_PULLUP_IOPORT (defined
 * above) where the 1.5k pullup resistor is connected. See description
 * above for details.
 */

/* --------------------------- Functional Range ---------------------------- */

#define USB_CFG_HAVE_INTRIN_ENDPOINT    1
/* Define this to 1 if you want to compile a version with two endpoints: The
 * default control endpoint 0 and an interrupt-in endpoint (any other endpoint
 * number).
 */
#define USB_CFG_HAVE_INTRIN_ENDPOINT3   0
/* Define this to 1 if you want to compile a version with three endpoints: The
 * default control endpoint 0, an interrupt-in endpoint 3 (or the number
 * configured below) and a catch-all default interrupt-in endpoint as above.
 * You must also define USB_CFG_HAVE_INTRIN_ENDPOINT to 1 for this feature.
 */
#define USB_CFG_EP3_NUMBER              3
/* If the so-called endpoint 3 is used, it can now be configured to any other
 * endpoint number (except 0) with this macro. Default if undefined is 3.
 */
/* #define USB_INITIAL_DATATOKEN           USBPID_DATA1 */
/* The above macro defines the startup condition for data toggling on the
 * interrupt/bulk endpoints 1 and 3. Defaults to USBPID_DATA1.
 * Since the token is toggled BEFORE sending any data, the first packet is
 * sent with the oposite value of this configuration!
 */
#define USB_CFG_IMPLEMENT_HALT          0
/* Define this to 1 if you also want to implement the ENDPOINT_HALT feature
 * for endpoint 1 (interrupt endpoint). Although you may not need this feature,
 * it is required by the standard. We have made it a config option because it
 * bloats the code considerably.
 */
#define USB_CFG_SUPPRESS_INTR_CODE      0
/* Define this to 1 if you want to declare interrupt-in endpoints, but don't
 * want to send any data over them. If this macro is defined to 1, functions
 * usbSetInterrupt() and usbSetInterrupt3() are omitted. This is useful if
 * you need the interrupt-in endpoints in order to comply to an interface
 * (e.g. HID), but never want to send any data. This option saves a couple
 * of bytes in flash memory and the transmit buffers in RAM.
 */
#define USB_CFG_INTR_POLL_INTERVAL      10
/* If you compile a version with endpoint 1 (interrupt-in), this is the poll
 * interval. The value is in milliseconds and must not be less than 10 ms for
 * low speed devices.
 */
#define USB_CFG_IS_SELF_POWERED         0
/* Define this to 1 if the device has its own power supply. Set it to 0 if the
 * device is powered from the USB bus.
 */
#define USB_CFG_MAX_BUS_POWER           100
/* Set this variable to the maximum USB bus power consumption of your device.
 * The value is in milliamperes. [It will be divided by two since USB
 * communicates power requirements in units of 2 mA.]
 */
#define USB_CFG_IMPLEMENT_FN_WRITE      0
/* Set this to 1 if you want usbFunctionWrite() to be called for control-out
 * transfers. Set it to 0 if you don't need it and want to save a couple of
 * bytes.
 */
#define USB_CFG_IMPLEMENT_FN_READ       0
/* Set this to 1 if you need to send control replies which are generated
 * "on the fly" when usbFunctionRead() is called. If you only want to send
 * data from a static buffer, set it to 0 and return the data from
 * usbFunctionSetup(). This saves a couple of bytes.
 */
#define USB_CFG_IMPLEMENT_FN_WRITEOUT   0
/* Define this to 1 if you want to use interrupt-out (or bulk out) endpoints.
 * You must implement the function usbFunctionWriteOut() which receives all
 * interrupt/bulk data sent to any endpoint other than 0. The endpoint number
 * can be found in 'usbRxToken'.
 */
#define USB_CFG_HAVE_FLOWCONTROL        0
/* Define this to 1 if you want flowcontrol over USB data. See the definition
 * of the macros usbDisableAllRequests() and usbEnableAllRequests() in
 * usbdrv.h.
 */
#define USB_CFG_LONG_TRANSFERS          0
/* Define this to 1 if you want to send/receive blocks of more than 254 bytes
 * in a single control-in or control-out transfer. Note that the capability
 * for long transfers increases the driver size.
 */
/* #define USB_RX_USER_HOOK(data, len)     if(usbRxToken == (uchar)USBPID_SETUP)
blinkLED(); */
/* This macro is a hook if you want to do unconventional things. If it is
 * defined, it's inserted at the beginning of received message processing.
 * If you eat the received message and don't want default processing to
 * proceed, do a return after doing your things. One possible application
 * (besides debugging) is to flash a status LED on each packet.
 */
/* #define USB_RESET_HOOK(resetStarts)     if(!resetStarts){hadUsbReset();} */
/* This macro is a hook if you need to know when an USB RESET occurs. It has
 * one parameter which distinguishes between the start of RESET state and its
 * end.
 */
/* #define USB_SET_ADDRESS_HOOK()              hadAddressAssigned(); */
/* This macro (if defined) is executed when a USB SET_ADDRESS request was
 * received.
 */
#define USB_COUNT_SOF                   0
/* define this macro to 1 if you need the global variable "usbSofCount" which
 * counts SOF packets. This feature requires that the hardware interrupt is
 * connected to D- instead of D+.
 */
/* #ifdef __ASSEMBLER__
 * macro myAssemblerMacro
 *     in      YL, TCNT0
 *     sts     timer0Snapshot, YL
 *     endm
 * #endif
 * #define USB_SOF_HOOK                    myAssemblerMacro
 * This macro (if defined) is executed in the assembler module when a
 * Start Of Frame condition is detected. It is recommended to define it to
 * the name of an assembler macro which is defined here as well so that more
 * than one assembler instruction can be used. The macro may use the register
 * YL and modify SREG. If it lasts longer than a couple of cycles, USB messages
 * immediately after an SOF pulse may be lost and must be retried by the host.
 * What can you do with this hook? Since the SOF signal occurs exactly every
 * 1 ms (unless the host is in sleep mode), you can use it to tune OSCCAL in
 * designs running on the internal RC oscillator.
 * Please note that Start Of Frame detection works only if D- is wired to the
 * interrupt, not D+. THIS IS DIFFERENT THAN MOST EXAMPLES!
 */
#define USB_CFG_CHECK_DATA_TOGGLING     0
/* define this macro to 1 if you want to filter out duplicate data packets
 * sent by the host. Duplicates occur only as a consequence of communication
 * errors, when the host does not receive an ACK. Please note that you need to
 * implement the filtering yourself in usbFunctionWriteOut() and
 * usbFunctionWrite(). Use the global usbCurrentDataToken and a static variable
 * for each control- and out-endpoint to check for duplicate packets.
 */
#define USB_CFG_HAVE_MEASURE_FRAME_LENGTH   0
/* define this macro to 1 if you want the function usbMeasureFrameLength()
 * compiled in. This function can be used to calibrate the AVR's RC oscillator.
 */
#define USB_USE_FAST_CRC                0
/* The assembler module has two implementations for the CRC algorithm. One is
 * faster, the other is smaller. This CRC routine is only used for transmitted
 * messages where timing is not critical. The faster routine needs 31 cycles
 * per byte while the smaller one needs 61 to 69 cycles. The faster routine
 * may be worth the 32 bytes bigger code size if you transmit lots of data and
 * run the AVR close to its limit.
 */

/* -------------------------- Device Description --------------------------- */

#define  USB_CFG_VENDOR_ID       0xc0, 0x16
/* USB vendor ID for the device, low byte first. If you have registered your
 * own Vendor ID, define it here. Otherwise you may use one of obdev's free
 * shared VID/PID pairs. Be sure to read USB-IDs-for-free.txt for rules!
 * *** IMPORTANT NOTE ***
 * This template uses obdev's shared VID/PID pair for Vendor Class devices
 * with libusb: 0x16c0/0x5dc.  Use this VID/PID pair ONLY if you understand
 * the implications!
 */
#define  USB_CFG_DEVICE_ID       0xdb, 0x27
/* This is the ID of the product, low byte first. It is interpreted in the
 * scope of the vendor ID. If you have registered your own VID with usb.org
 * or if you have licensed a PID from somebody else, define it here. Otherwise
 * you may use one of obdev's free shared VID/PID pairs. See the file
 * USB-IDs-for-free.txt for details!
 * *** IMPORTANT NOTE ***
 * This template uses obdev's shared VID/PID pair for Vendor Class devices
 * with libusb: 0x16c0/0x5dc.  Use this VID/PID pair ONLY if you understand
 * the implications!
 */
#define USB_CFG_DEVICE_VERSION  0x00, 0x01
/* Version number of the device: Minor number first, then major number.
 */
#define USB_CFG_VENDOR_NAME
'i','t','a','p','a','r','c','a','d','e','.','c','o','m'
#define USB_CFG_VENDOR_NAME_LEN 14
/* These two values define the vendor name returned by the USB device. The name
 * must be given as a list of characters under single quotes. The characters
 * are interpreted as Unicode (UTF-16) entities.
 * If you don't want a vendor name string, undefine these macros.
 * ALWAYS define a vendor name containing your Internet domain name if you use
 * obdev's free shared VID/PID pair. See the file USB-IDs-for-free.txt for
 * details.
 */
#define USB_CFG_DEVICE_NAME     'D','u','o',' ','P','o','p',' ','U','S','B'
#define USB_CFG_DEVICE_NAME_LEN 11
/* Same as above for the device name. If you don't want a device name, undefine
 * the macros. See the file USB-IDs-for-free.txt before you assign a name if
 * you use a shared VID/PID.
 */
/*#define USB_CFG_SERIAL_NUMBER   'N', 'o', 'n', 'e' */
/*#define USB_CFG_SERIAL_NUMBER_LEN   0 */
/* Same as above for the serial number. If you don't want a serial number,
 * undefine the macros.
 * It may be useful to provide the serial number through other means than at
 * compile time. See the section about descriptor properties below for how
 * to fine tune control over USB descriptors such as the string descriptor
 * for the serial number.
 */
#define USB_CFG_DEVICE_CLASS        0    /* set to 0 if deferred to interface */
#define USB_CFG_DEVICE_SUBCLASS     0
/* See USB specification if you want to conform to an existing device class.
 * Class 0xff is "vendor specific".
 */
#define USB_CFG_INTERFACE_CLASS     0x03  /* HID */ /* define class here if not at
device level */
#define USB_CFG_INTERFACE_SUBCLASS  0
#define USB_CFG_INTERFACE_PROTOCOL  0
/* See USB specification if you want to conform to an existing device class or
 * protocol. The following classes must be set at interface level:
 * HID class is 3, no subclass and protocol required (but may be useful!)
 * CDC class is 2, use subclass 2 and protocol 1 for ACM
 */
#define USB_CFG_HID_REPORT_DESCRIPTOR_LENGTH    35
/* Define this to the length of the HID report descriptor, if you implement
 * an HID device. Otherwise don't define it or define it to 0.
 * If you use this define, you must add a PROGMEM character array named
 * "usbHidReportDescriptor" to your code which contains the report descriptor.
 * Don't forget to keep the array and this define in sync!
 */

/* #define USB_PUBLIC static */
/* Use the define above if you #include usbdrv.c instead of linking against it.
 * This technique saves a couple of bytes in flash memory.
 */

/* ------------------- Fine Control over USB Descriptors ------------------- */
/* If you don't want to use the driver's default USB descriptors, you can
 * provide our own. These can be provided as (1) fixed length static data in
 * flash memory, (2) fixed length static data in RAM or (3) dynamically at
 * runtime in the function usbFunctionDescriptor(). See usbdrv.h for more
 * information about this function.
 * Descriptor handling is configured through the descriptor's properties. If
 * no properties are defined or if they are 0, the default descriptor is used.
 * Possible properties are:
 *   + USB_PROP_IS_DYNAMIC: The data for the descriptor should be fetched
 *     at runtime via usbFunctionDescriptor(). If the usbMsgPtr mechanism is
 *     used, the data is in FLASH by default. Add property USB_PROP_IS_RAM if
 *     you want RAM pointers.
 *   + USB_PROP_IS_RAM: The data returned by usbFunctionDescriptor() or found
 *     in static memory is in RAM, not in flash memory.
 *   + USB_PROP_LENGTH(len): If the data is in static memory (RAM or flash),
 *     the driver must know the descriptor's length. The descriptor itself is
 *     found at the address of a well known identifier (see below).
 * List of static descriptor names (must be declared PROGMEM if in flash):
 *   char usbDescriptorDevice[];
 *   char usbDescriptorConfiguration[];
 *   char usbDescriptorHidReport[];
 *   char usbDescriptorString0[];
 *   int usbDescriptorStringVendor[];
 *   int usbDescriptorStringDevice[];
 *   int usbDescriptorStringSerialNumber[];
 * Other descriptors can't be provided statically, they must be provided
 * dynamically at runtime.
 *
 * Descriptor properties are or-ed or added together, e.g.:
 * #define USB_CFG_DESCR_PROPS_DEVICE   (USB_PROP_IS_RAM | USB_PROP_LENGTH(18))
 *
 * The following descriptors are defined:
 *   USB_CFG_DESCR_PROPS_DEVICE
 *   USB_CFG_DESCR_PROPS_CONFIGURATION
 *   USB_CFG_DESCR_PROPS_STRINGS
 *   USB_CFG_DESCR_PROPS_STRING_0
 *   USB_CFG_DESCR_PROPS_STRING_VENDOR
 *   USB_CFG_DESCR_PROPS_STRING_PRODUCT
 *   USB_CFG_DESCR_PROPS_STRING_SERIAL_NUMBER
 *   USB_CFG_DESCR_PROPS_HID
 *   USB_CFG_DESCR_PROPS_HID_REPORT
 *   USB_CFG_DESCR_PROPS_UNKNOWN (for all descriptors not handled by the driver)
 *
 * Note about string descriptors: String descriptors are not just strings, they
 * are Unicode strings prefixed with a 2 byte header. Example:
 * int  serialNumberDescriptor[] = {
 *     USB_STRING_DESCRIPTOR_HEADER(6),
 *     'S', 'e', 'r', 'i', 'a', 'l'
 * };
 */

#define USB_CFG_DESCR_PROPS_DEVICE                  0
#define USB_CFG_DESCR_PROPS_CONFIGURATION           0
#define USB_CFG_DESCR_PROPS_STRINGS                 0
#define USB_CFG_DESCR_PROPS_STRING_0                0
#define USB_CFG_DESCR_PROPS_STRING_VENDOR           0
#define USB_CFG_DESCR_PROPS_STRING_PRODUCT          0
#define USB_CFG_DESCR_PROPS_STRING_SERIAL_NUMBER    0
#define USB_CFG_DESCR_PROPS_HID                     0
#define USB_CFG_DESCR_PROPS_HID_REPORT              0
#define USB_CFG_DESCR_PROPS_UNKNOWN                 0

/* ----------------------- Optional MCU Description ------------------------ */

/* The following configurations have working defaults in usbdrv.h. You
 * usually don't need to set them explicitly. Only if you want to run
 * the driver on a device which is not yet supported or with a compiler
 * which is not fully supported (such as IAR C) or if you use a differnt
 * interrupt than INT0, you may have to define some of these.
 */
/* #define USB_INTR_CFG            MCUCR */
/* #define USB_INTR_CFG_SET        ((1 &lt;&lt; ISC00) | (1 &lt;&lt; ISC01)) */
/* #define USB_INTR_CFG_CLR        0 */
/* #define USB_INTR_ENABLE         GIMSK */
/* #define USB_INTR_ENABLE_BIT     INT0 */
/* #define USB_INTR_PENDING        GIFR */
/* #define USB_INTR_PENDING_BIT    INTF0 */
/* #define USB_INTR_VECTOR         SIG_INTERRUPT0 */
```

```
#define USB_INTR_VECTOR INT0_vect

#endif /* __usbconfig_h_included__ */
```

To help test if your Pro Trinket is acting as a USB keyboard, I have included the following test keyboard sketch below. Pulling pin 4 low will cause the LED on the Pro Trinket to turn off as it generates 'hello world' with a carriage return multiple times on the screen.

```
#include "UsbKeyboard.h"

#define BUTTON_PIN     4
#define ledPin         13

void setup() {
  pinMode(BUTTON_PIN, INPUT);
      // ...with a pullup
  digitalWrite(BUTTON_PIN, HIGH);

  pinMode(ledPin, OUTPUT);
  digitalWrite(ledPin, LOW);


   // Disable timer0 since it can mess with the USB timing. Note that
  // this means some functions such as delay() will no longer work.
  TIMSK0&amp;=!(1&lt;&lt;TOIE0);

  // Clear interrupts while performing time-critical operations
  cli();

  // Force re-enumeration so the host will detect us
  usbDeviceDisconnect();
  delayMs(250);
  usbDeviceConnect();

  // Set interrupts again
  sei();

}

void loop() {

UsbKeyboard.update();

digitalWrite(ledPin,HIGH);

if (digitalRead(BUTTON_PIN) == 0) {
    digitalWrite(ledPin,LOW);
    UsbKeyboard.sendKeyStroke(KEY_H);
    UsbKeyboard.sendKeyStroke(KEY_E);
    UsbKeyboard.sendKeyStroke(KEY_L);
    UsbKeyboard.sendKeyStroke(KEY_L);
    UsbKeyboard.sendKeyStroke(KEY_O);

    UsbKeyboard.sendKeyStroke(KEY_SPACE);

    UsbKeyboard.sendKeyStroke(KEY_W);
    UsbKeyboard.sendKeyStroke(KEY_O);
    UsbKeyboard.sendKeyStroke(KEY_R);
    UsbKeyboard.sendKeyStroke(KEY_L);
    UsbKeyboard.sendKeyStroke(KEY_D);

    UsbKeyboard.sendKeyStroke(KEY_ENTER);
```

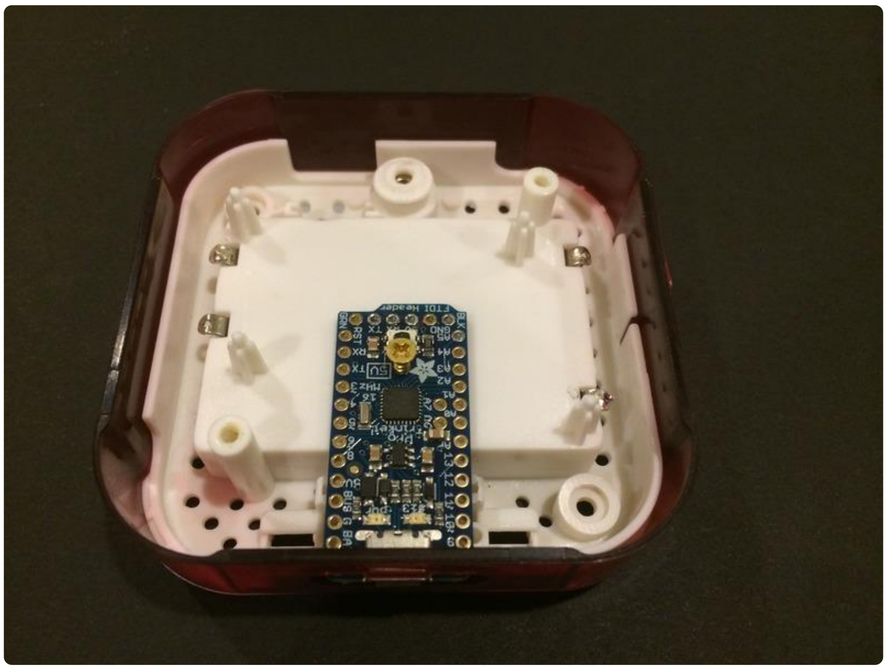
```
     delayMs(20);
    }

}

void delayMs(unsigned int ms) {
   /*
  */
  for (int i = 0; i &lt; ms; i++) {
    delayMicroseconds(1000);
  }
}
```

This allows you to create a virtual USB keyboard on the Pro Trinket. For more information about V-USB visit the V-USB site ().

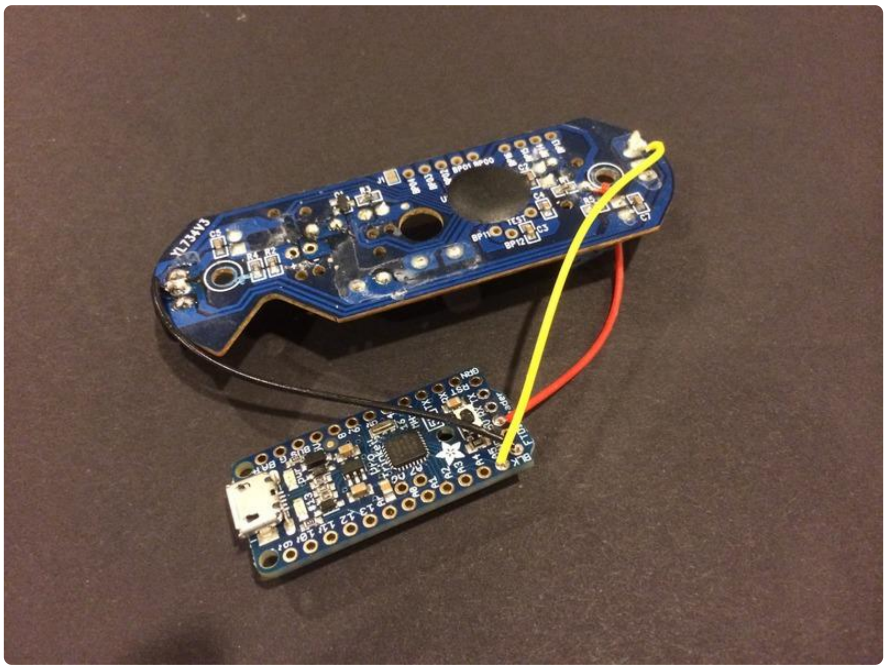
Build

Now that we have the Arduino code needed to translate the popper IR transmissions into USB keyboard responses, it was time to modify the Duo Pop IR receiver box so that the Pro Trinket could use the existing IR sensors of the Duo Pop receiver while providng a USB out connection from the Duo Pop box.

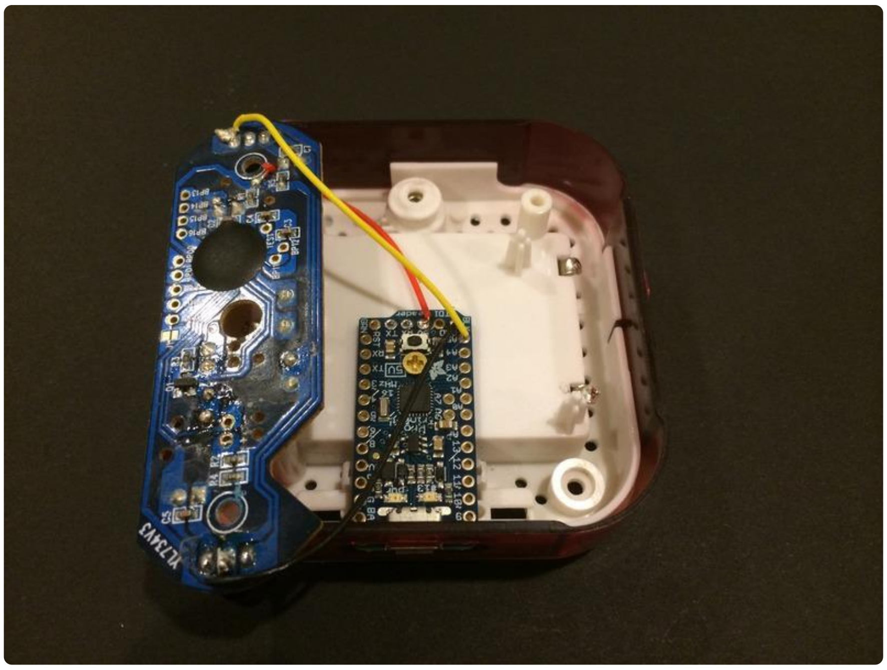
Step One: Disassemble Duo Pop IR receiver and desolder receiver IR board. Cut holding mounts (using diagonal cutters) that held the power pushbutton and led in place so you can place the Pro Trinket in the location of the power switch. Drill mounting hole for Pro Trinket as shown below. A 1/16 drill bit is used to drill the hole and a #2 x 3/8 inch mounting screw is used to make sure the Pro Trinket stays secure.



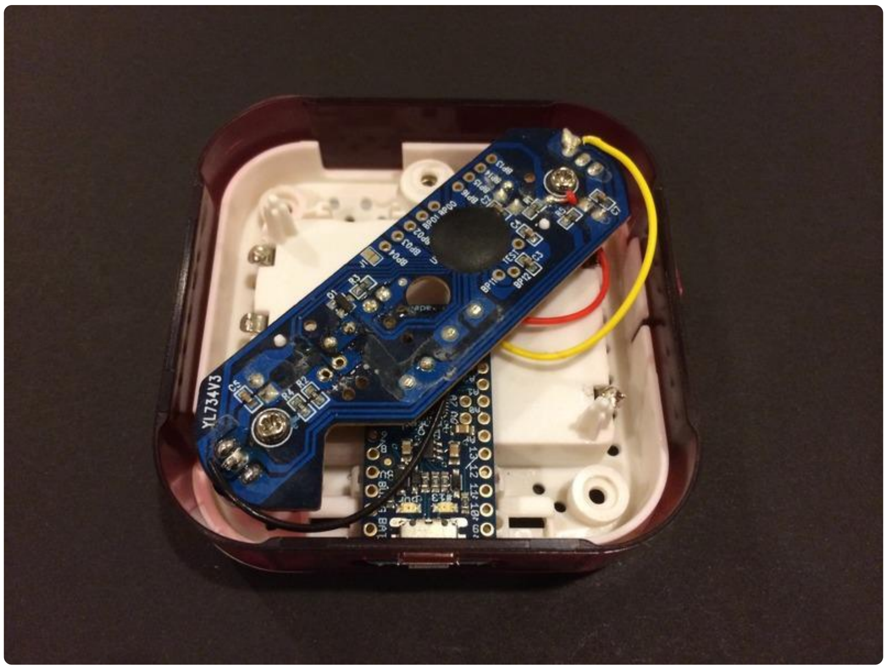
Step Two: Rewire and solder the IR sensor board wiring to the Pro Trinket as shown in image below. The only wire that remains connected to the original IR sensor board is the red wire. All other wires are removed, rearranged or resoldered to the original IR sensor board.



Step Three: Remount the Pro Trinket to the case.



Step Four: Remount the IR module board to the case. Your IR to USB receiver should look similar to the picture below.



Step 5: Reassemble remaining portions of the case so you can program your newly built Duo Pop IR USB receiver module.

Program

Combining code from both the IR decoding and creating a virtual USB keyboard learning sections provides the core pieces of code to create the Pro Trinket Dup Pop IR to USB reciever.   A final listing of code using these core pieces is shown below.

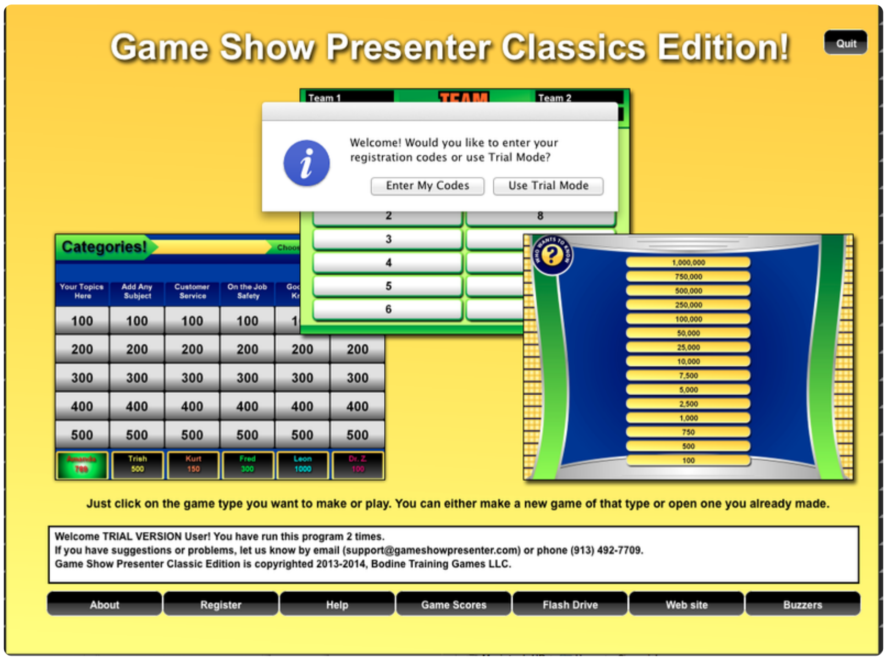
The trick in making sure the functionality of the IR decoding and the USB keyboard worked together was to make sure the necessary polling associated with the USB keyboard was located within the proper IR scanning loop.

```
/* Pro Trinket Duo Pop Receiver IR decoder sketch!

 This sketch/program uses the Arduino and a Duo Pop Receiver IR board to
 decode IR received data from 4 of the Poppers and translate the received data
 into keystrokes. This can be used to make a IR receiver
 (by looking for a particular code)
 or transmitter (by pulsing an IR LED at ~38KHz for the
 durations detected.  This sketch leveraged tutorials from www.ladyada.net and
adafruit.com.

 Created by: www.itaparcade.com
 Twitter: @iTapArcade
 e-mail: info@itaparcade.com
 Date created: 9/20/2014
 Date updated: 9/20/2014

 For preassembled Duo Pop kit information, contact us at info@itaparcade.com

 */
//#include &lt;SendOnlySoftwareSerial.h&gt;
#include "UsbKeyboard.h"
```

```
// We need to use the 'raw' pin reading methods
// because timing is very important here and the digitalRead()
// procedure is slower!
//uint8_t IRpin = 2;
// Digital pin #2 is the same as Pin D2 see
// http://arduino.cc/en/Hacking/PinMapping168 for the 'raw' pin mapping
#define IRpin_PIN      PINC
#define IRpin          PINC5  // Connect to A5 on Pro Trinket

#define ledPin         13

// the maximum pulse we'll listen for - 1 milliseconds based on Duo Pop pulses

#define MAXPULSE 1000

// what our timing resolution should be, larger is better
// as its more 'precise' - but too large and you wont get
// accurate timing
#define RESOLUTION 20

#define FUZZINESS 45

// we will store up to 100 pulse pairs (this is -a lot-)
uint16_t pulses[100][2];  // pair is high and low pulse
uint8_t currentpulse = 0; // index for pulses we're storing

// Red Code

int DUOPOPsignalRed[] = { // ON, OFF (in 10's of microseconds)
140, 50,
100, 50,
100, 100,
50, 50,
100, 50,
100, 50,
100, 100,
50, 50,
100, 50,
100};

int DUOPOPsignalBlue[] = { // ON, OFF (in 10's of microseconds)
140, 50,
100, 50,
100, 50,
100, 100,
50, 50,
100, 50,
100, 50,
100, 100,
50, 50,
100};


int DUOPOPsignalYellow[] = { // ON, OFF (in 10's of microseconds)
140,
100, 50,
50, 100,
100, 50,
50, 100,
100, 50,
50, 100,
100, 50,
50, 100,
50, 100};

int DUOPOPsignalGreen[] = { // ON, OFF (in 10's of microseconds)
140,
50, 100,
100, 50,
50, 100,
100, 50,
50, 100,
100, 50,
50, 100,
100, 50,
50, 100};

//SendOnlySoftwareSerial Serial(0);

void setup(void) {
    // initialize the LED pin as an output.
 // Serial.begin(9600);
 // Serial.println("Ready to decode IR!");
  pinMode(ledPin, OUTPUT);
  digitalWrite(ledPin, LOW);
  pinMode(IRpin, INPUT);
//  pinMode(Pwrpin, OUTPUT);
//  digitalWrite (Pwrpin, HIGH);
 // Serial.begin(9600);
 // Serial.println("Ready to decode IR!");


   // Disable timer0 since it can mess with the USB timing. Note that
  // this means some functions such as delay() will no longer work.
  TIMSK0&amp;=!(1&lt;&lt;TOIE0);

  // Clear interrupts while performing time-critical operations
  cli();

  // Force re-enumeration so the host will detect us
  usbDeviceDisconnect();
  delayMs(250);
  usbDeviceConnect();

  // Set interrupts again
  sei();
}

// What percent we will allow in variation to match the same code \\ #define
FUZZINESS 20

void loop(void) {

  int numberpulses;

  numberpulses = listenForIR();

//  Serial.print("Heard ");
//  Serial.print(numberpulses);
//  Serial.println("-pulse long IR signal");
  if (numberpulses == 10)
  {
   // Serial.println("Received IR!");
   // digitalWrite(LED, HIGH);
   //   delay(50);
    //  digitalWrite(LED, LOW);
  if (IRcompare(numberpulses, DUOPOPsignalRed)) {
  //  Serial.println("RED Buzzed In");
 // UsbKeyboard.sendKeyStroke(KEY_A);
//    digitalWrite(ledPin, LOW); // Toggle status LED
  //  delayMs(500);
    UsbKeyboard.sendKeyStroke(KEY_1);
    digitalWrite(ledPin, HIGH); // Toggle status LED
    delayMs(500);
  }
  if (IRcompare(numberpulses, DUOPOPsignalBlue)) {
   // Serial.println("BLUE Buzzed In");
  UsbKeyboard.sendKeyStroke(KEY_3);
     digitalWrite(ledPin, HIGH); // Toggle status LED
    delayMs(500);
  }
    if (IRcompare(numberpulses, DUOPOPsignalYellow)) {
   // Serial.println("Yellow Buzzed In");
     UsbKeyboard.sendKeyStroke(KEY_2);
     digitalWrite(ledPin, HIGH); // Toggle status LED
    delayMs(500);

  }
   if (IRcompare(numberpulses, DUOPOPsignalGreen)) {
   // Serial.println("Green Buzzed In");
     UsbKeyboard.sendKeyStroke(KEY_4);
     digitalWrite(ledPin, HIGH); // Toggle status LED
    delayMs(500);

  }
  }
  // Show Trigger Status via LED
digitalWrite(ledPin, LOW);

}

int listenForIR(void) {
  currentpulse = 0;

  while (1) {

    uint16_t highpulse, lowpulse;  // temporary storage timing
    highpulse = lowpulse = 0; // start out with no pulse length
//  while (digitalRead(IRpin)) { // this is too slow!
   // while (IRpin_PIN &amp; (1 &lt;&lt; IRpin)) {
     while (IRpin_PIN &amp; _BV(IRpin)) {

       // Need to poll USB periodically while waiting for popper responses
       UsbKeyboard.update();

       // count off another few microseconds
       highpulse++;
       delayMicroseconds(RESOLUTION);

       // If the pulse is too long, we 'timed out' - either nothing
       // was received or the code is finished, so print what
       // we've grabbed so far, and then reset
       if (((highpulse &gt;= MAXPULSE) &amp;&amp; (currentpulse != 0)) ||
(currentpulse == 10)) {
         return currentpulse;
       }
    }
    // we didn't time out so lets stash the reading
    pulses[currentpulse][0] = highpulse;

    // same as above
    while (! (IRpin_PIN &amp; _BV(IRpin))) {
       // pin is still LOW
       lowpulse++;
       delayMicroseconds(RESOLUTION);
       if (((lowpulse &gt;= MAXPULSE)  &amp;&amp; (currentpulse != 0)) ||
(currentpulse == 10)) {
         return currentpulse;
       }
    }
    pulses[currentpulse][1] = lowpulse;

    // we read one high-low pulse successfully, continue!
    currentpulse++;
  }
```

```
}


boolean IRcompare(int numpulses, int Signal[]) {

  for (int i=0; i&lt; numpulses-1; i++) {
    int oncode = pulses[i][1] * RESOLUTION / 10;
    int offcode = pulses[i+1][0] * RESOLUTION / 10;


 //   Serial.print(oncode); // the ON signal we heard
 //   Serial.print(" - ");
 //   Serial.print(Signal[i*2 + 0]); // the ON signal we want


    // check to make sure the error is less than FUZZINESS percent
    if ( abs(oncode - Signal[i*2 + 0]) &lt;= (Signal[i*2 + 0] * FUZZINESS / 100)) {
  //    Serial.print(" (ok)");


    } else {
  //    Serial.print(" (x)");
      // we didn't match perfectly, return a false match
      return false;
    }


 //   Serial.print("  \t"); // tab
 //   Serial.print(offcode); // the OFF signal we heard
 //   Serial.print(" - ");
 //   Serial.print(Signal[i*2 + 1]); // the OFF signal we want


    if ( abs(offcode - Signal[i*2 + 1]) &lt;= (Signal[i*2 + 1] * FUZZINESS / 100)) {
  //    Serial.print(" (ok)");
    } else {
  //    Serial.print(" (x)");
      // we didn't match perfectly, return a false match
      return false;
    }

   // Serial.println();
  }
  // Everything matched!
  return true;
}

/**
 * Define our own delay function so that we don't have to rely on
 * operation of timer0, the interrupt used by the internal delay()
 */
void delayMs(unsigned int ms)
{
  for (int i = 0; i &lt; ms; i++) {
    delayMicroseconds(1000);
  }
}
```

Play



Welcome! Would you like to enter your registration codes or use Trial Mode?

Just click on the game type you want to make or play. You can either make a new game of that type or open one you already made.

Welcome TRIAL VERSION User! You have run this program 2 times.
If you have suggestions or problems, let us know by email (support@gameshowpresenter.com) or phone (913) 492-7709.
Game Show Presenter Classic Edition is copyrighted 2013-2014, Bodine Training Games LLC.

With the Duo Pop Popper USB receiver built based on the Pro Trinket, it is now time to install the Game Show Software on your PC or Mac and start using the poppers for play. The figure above shows the game show presenter classic software we used on the Mac to test our USB receiver. We have also included a You Tube video (below) of the game show presenter software being used with the Duo Pop receiver. This video was created by the Game Show Presenter developer so you can see how the USB receiver interacts with the game show software.

We first tested our modified IR USB receiver to make sure it was receiving and sending the corresponding keystroke based on the popper color. Using the TextEdit program on our Mac laptop, we could see keystrokes being generated as we pressed the different poppers. We programmed the Pro Trinket LED to light when it received IR transmissions from the poppers. This shows up as a red glow within the Duo Pop receiver case as shown in the picture. When initially plugging in the USB cable, the Pro Trinket is delayed emulating a USB keyboard due to the Bootloader. In this mode, it flashes red letting you know you can upload a new sketch if desired. After this mode, it should stop flashing and be ready to receive popper responses.

We programmed the red popper to generate a 1 keystroke, the yellow popper to generate a 2 keystroke, the blue popper to generate a 3 keystroke, and the green popper to generate a 4 keystroke. We then started the Game Show Presenter Software and validated responses from the poppers. We have included a game show presenter template below that you can load to test your poppers once building the receiver. The team names are based on the popper colors in our template.

iTapArcadeDemo.gspc

There should be enough range on the poppers to use them in a small classroom setting if you can split the classroom into different groups. Have a designated person from each group assigned to press the popper if that particular group wants to answer based on the game. You can also just have four students at a time use the poppers for small group competitions. This project is also great for a game show night at house parties (especially around the holidays). Either way, you now have an inexpensive wireless game show popper system for keeping students engaged in the classroom or at home! Have fun with IR popper decoding and V-USB programming using the Arduino and Pro Trinket! Follow us on Twitter if you like these type of projects @iTapArcade and make sure to follow @adafruit for their excellent tutorials!