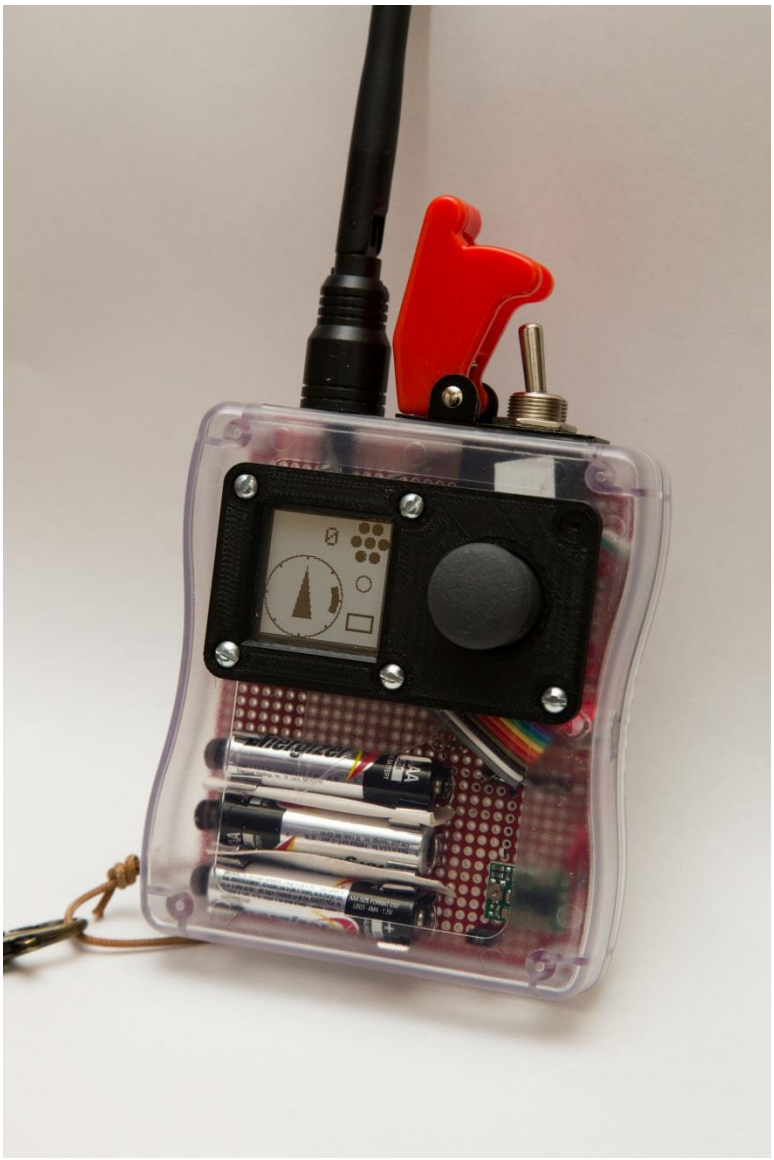
An Arduino Based R/C KAP Controller

David Wheeler

dwheeler20874@gmail.com



Introduction

Taking inspiration from repackaged R/C transmitters by Cris Benton and David Mitchell, I decided last summer to build an R/C system for my own KAP rig. Like Cris and David, I started with an off-the-shelf R/C transmitter and receiver. The Hobby King T6A was my system of choice. The receiver was used as-is, but the transmitter I modified beyond recognition. This paper describes the new transmitter module and how it works.

The description here doesn't provide step by step instructions. But it is intended as a guide for any technically-minded individual who wants to develop a similar system.

This design, as mentioned above, is based on an available R/C system. But only the 2.4 GHz RF module (and antenna) from the original transmitter is retained. An Arduino processor feeds the RF module with a PPM signal, telling it where to position each of the servos. The operator controls the system through a user interface consisting of a thumb joystick and an LCD. The Arduino interfaces with these components, translates the operator's intentions into servo settings, and transmits them to the rig through the RF module.

The rest of this document is organized into four topics: requirements, functionality, system components and theory of operation. The requirements section briefly introduces what the system is intended to do. Then the functionality section gives an operators view of the transmitter, describing how it is used to perform KAP operations. The system components section describes the construction of this transmitter. And, finally, a set of appendices discuss some key technical aspects of this project.

Discussion

A discussion topic on this R/C system was created on Cris Benton's KAP Forum. If you have any questions about this system, or are interested in further information, that would be the place to go.

http://arch.ced.berkeley.edu/kap/discuss/index.php?p=/discussion/5325

Online Repository

This document, along with the source for the Arduino, is available via Git:

https://github.com/howtokap/kap-tx1

Requirements

Over the years, I've made numerous attempts to develop different R/C KAP systems. So I have a lot of ideas of what such a system could do. Rather than design a do-everything system, though, I decided to minimize the functionality of this system. This, I hoped, would improve my odds of success this time. These were the basic functional requirements for this system:

- Set pan (full 360 degree range),
- Set tilt (+135 to -45 degrees from vertical)
- Trigger the shutter.

- Indicators should show the current pan/tilt orientation.
- Operable with one hand. (Preferable while wearing a glove.)
- Small enough to carry on a neck strap (so I could drop it when necessary.)

If I could get that much working, I'd have a usable system. Additional functionality could be added later.

Functionality

Here's how the system works from an operator's perspective.

Manual Pan Control

The pan angle is adjusted by moving the thumb joystick left or right. "Bumping" the knob left or right momentarily adjusts the pan angle by one step of 15 degrees. If the joystick is held to the left or right, the pan angle will make repeated steps.

Manual Tilt Control

The tilt angle is adjusted by moving the thumb joystick up or down. As with pan, bumps adjust the angle in single steps, holding the control results in repeated steps. Tilt adjustments are made in 15 degree increments.

Motion Smoothing

The Arduino module translates the operator's pan and tilt selections into servo positions. The software is calibrated so that the actual rig orientation matches the pan and tilt shown on the display. With this feature, the operator can read the rig's orientation from the display even if the rig itself can't be seen clearly.

When the Arduino changes the pan and tilt settings, it uses a constant acceleration to start and stop the servo motion. This prevents jerky movements that could induce unwanted swinging or vibration in the rig.

The Arduino also manages the rotation range of the pan servo. As the operator adjusts the pan angle continuously in one direction, the processor recognizes when the pan servo is saturated and it needs to "rewind" to reach the indicated heading.

Manual Shutter Control

The camera shutter is controlled via a servo signal using a GentLED infrared trigger. Whenever the operator presses the joystick button, a shutter press is triggered. This shutter press may be delayed, however, if the rig is in motion. The Arduino waits until any prior motion is completed and a brief stabilization period has passed before it triggers the camera.

Future: Shooting Sequences

It should be straightforward to implement alternative shooting modes with the Arduino software. What has been described above would be single-shot mode. A cluster shooting mode would add the capability to shoot multiple exposures with one button press. For example, the system could take the

first shot with the operator's intended pan and tilt, then take 8 more shots varying the pan and tilt angles slightly for these additional exposures. This would essentially automate the process of "compositional bracketing" as Cris Benton calls it.

Other sequential shooting modes could be used to produce horizontal or vertical panoramic images.

Future: AutoKAP Modes

Beyond sequential shooting modes, this control system could implement a full AutoKAP program. When put in AutoKAP mode, it would start shooting automatically using pre-programmed pan and tilt coordinates without operator intervention.

When the operator wants to return to intentional shooting, he or she could switch the system back to single-shot mode.

System Components

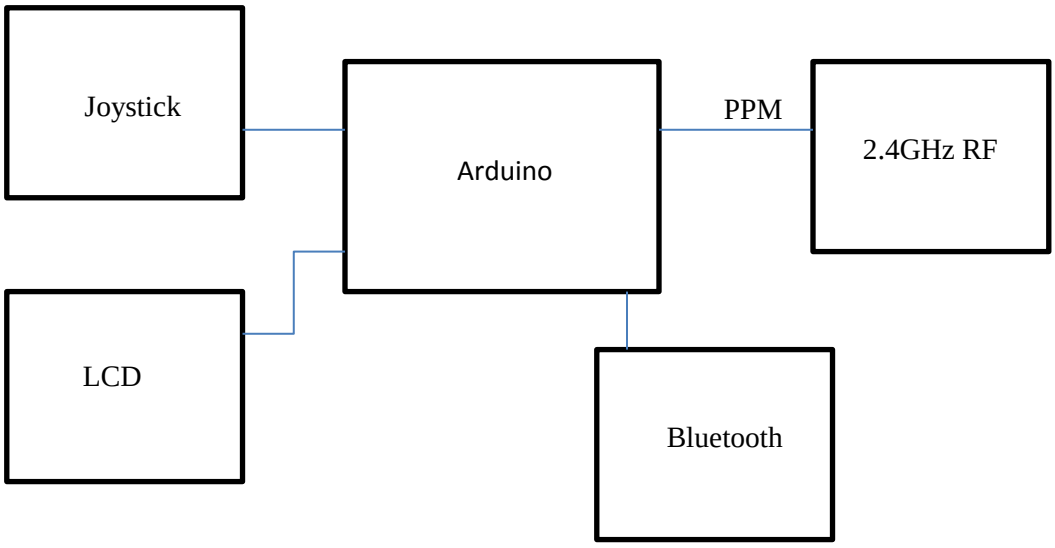
The transmitter system is designed with an Arduino processor at the. It has a thumb joystick to provide user input and an LCD module for output. The Arduino interfaces with the transmitter module from the HK-T6A to achieve its radio function. Finally, an on/off switch, voltage regulator and a set of AAA batteries provide the power. Most of the components, as well as the case, were ordered from Sparkfun

A more detailed Bill of Materials is included here:

Component	Source	Part Number
Arduino Pro Micro – 5V/16MHz	Sparkfun.com	https://www.sparkfun.com/products/12640 DEV-12640
RF module	Hobby King T6A	N/A
Antenna	Hobby King T6A	N/A
Thumb Joystick and Breakout Board	Sparkfun.com	https://www.sparkfun.com/products/9032 https://www.sparkfun.com/products/9110
SHARP LCD Module and Breakout board	adafuit	http://www.adafruit.com/product/1393
Adjustable Voltage Regulator	Pololu	https://www.pololu.com/product/2118
Bluetooth module	Sparkfun.com	https://www.sparkfun.com/products/12576
Power switch and cover	Sparkfun.com	https://www.sparkfun.com/products/9276 https://www.sparkfun.com/products/9278
Prototype Board	Sparkfun.com	https://www.sparkfun.com/products/8619
Case	Sparkfun.com	https://www.sparkfun.com/products/8632

Block Diagram

As this diagram shows, the Arduino processor is at the center of the system architecture. It works with the user interface components, the joystick and LCD, and sends servo positions to the 2.4GHz RF module. The bluetooth component is shown but this is not used, currently.



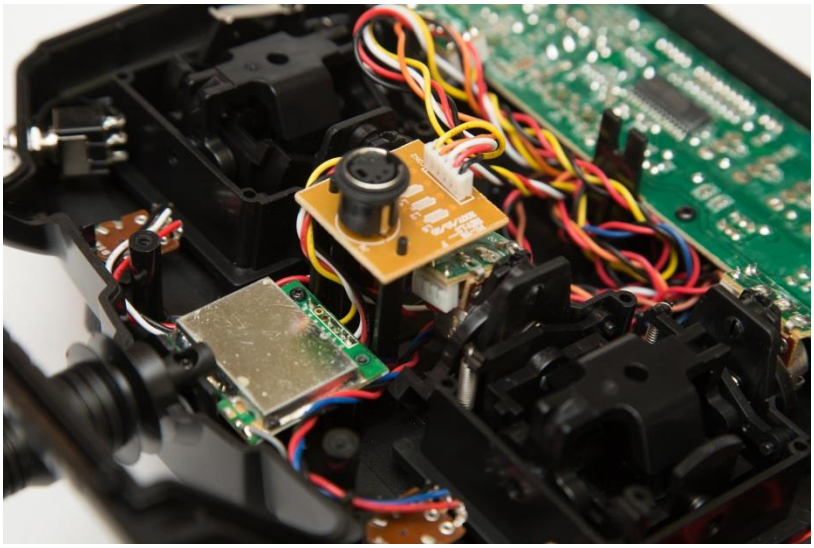
Arduino Pro Micro 5V/16MHz

The Arduino Pro Micro from Sparkfun is based on the ATmega32U4. It is quite small, making it more attractive than the standard arduino for this project. The dedicated timers of this processor support generating the key PPM signal. And the other pins and functions proved to be a perfect match for this project.

RF Module and Antenna

These were components taken from the Hobby King T6A transmitter.

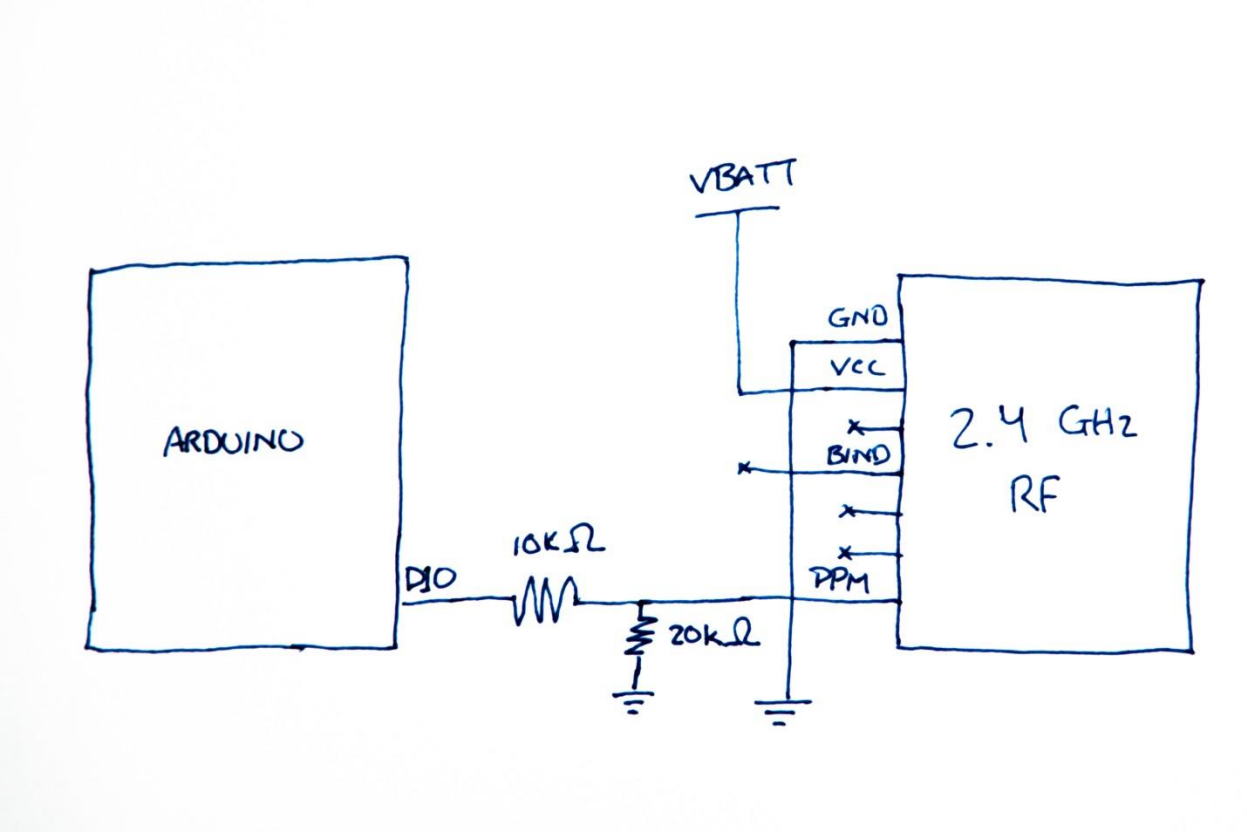
WARNING: When disassembling the Hobby King transmitter, be very careful not to cut or damage the wire connecting the RF module to the antenna! This "wire" is actually a tiny coaxial cable and would be difficult to repair if cut. To free the antenna from the case, I had to cut the case with a dremel tool.



The connections on the RF module are:

- Ground
- VCC (>4 volts)
- no connection
- Bind (white)
- no connection
- no connection
- PPM (yellow)

The connections between the Arduino and RF module are as shown here:



The RF module contains its own 3.3V LDO voltage regulator so the VCC input voltage can be any voltage above about 4 volts. I connected the battery (nominally 4.5 V) directly to the transmitter rather than feeding it from one of the other voltage regulators in my system.

The “bind” input should be connected through a momentary switch to ground. But in the current implementation, it is not connected. The binding function is activated by shorting that terminal while power is applied. (I was hoping to activating binding with a digital signal from the Arduino but didn’t succeed. The Tx module only seems to look for the bind signal immediately after power-up.)

The PPM (Pulse Position Modulation) input is the key to the whole project. This signal describes where the six servos should be positioned. The RF module transmits this information to the receiver where it is translated into the PWM servo signals. The detailed specifications of the PPM signal are as follows:

- Format: normally high with low pulses.
- Pulse duration: 400uS
- Frame frequency: 50Hz (20mS)
- Channels: 6 (denoted by 7 PPM pulses)
- Nominal inter-pulse interval: 1-2ms
- Min/Max pulse interval: 0.7ms – 2.3 ms.

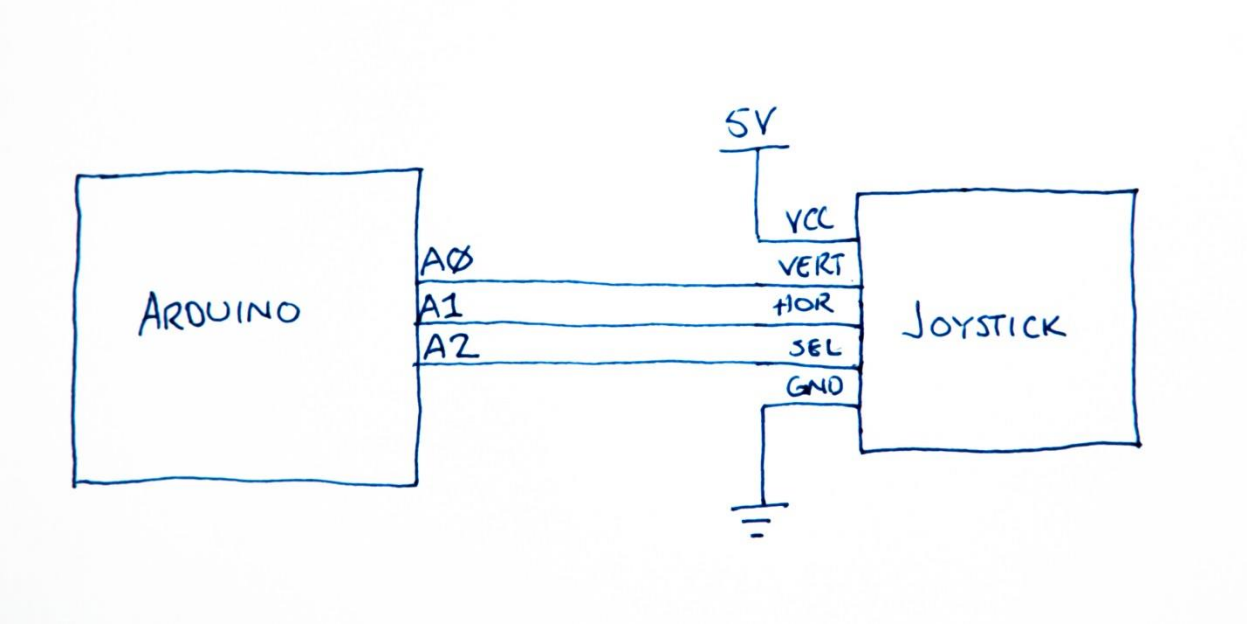
More information on this signal and how it is generated can be found below under Theory of Operation.

Thumb Joystick

The thumb joystick provides potentiomers for X and Y input. These are read with two of the available A/D channels on the Arduino.

The joystick also features a momentary switch when the stick is pressed down. This is used for the shutter function.

The following diagram shows the connection of the joystick to the Arduino:



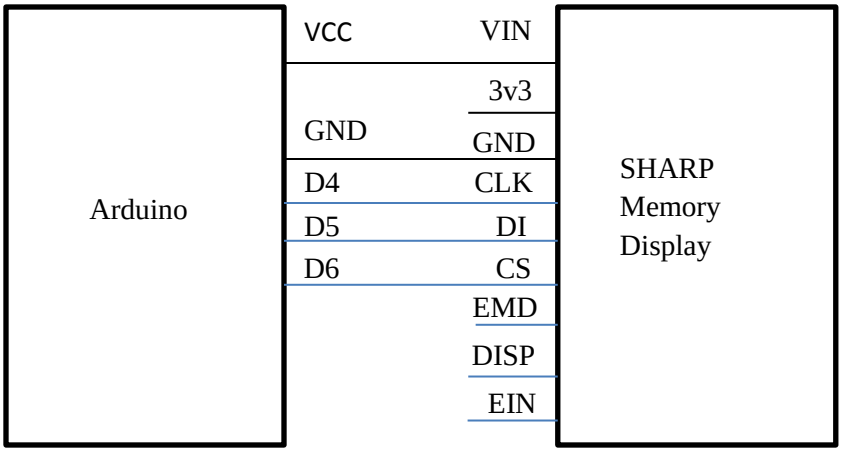
[Update: Joystick SEL now connects to Arduino D9]

Having pan, tilt and shutter all on the joystick enables operating all the primary functions with one finger, which is ideal. The large-ish knob also seems quite usable with gloves on. I'm hoping field testing proves this out.

LCD Module

The LCD display provides visual feedback on the pan and tilt settings. It also shows the shooting mode, shutter state, horizontal/vertical indicator and an auto/manual indicator.

The Adafruit SHARP Memory Display Breakout (http://www.adafruit.com/product/1393) was used as it provides a sunlight readable display.



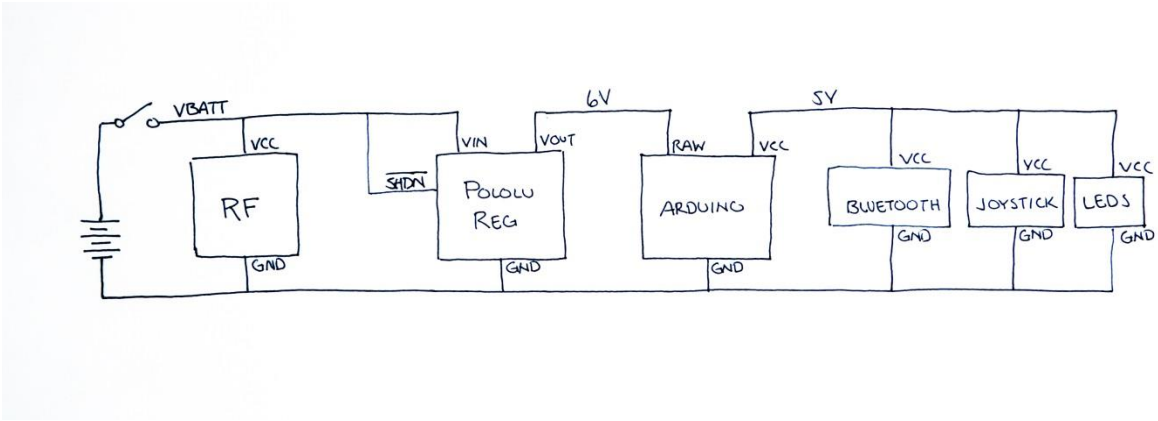
The LCD module was programmed using the Adafruit GFX library.

Power Switch

The main power switch resides on the top of the case and is protected by a missile switch cover. The switch cover may add some degree of ruggedness and prevent the unit from turning on accidentally in my equipment bag. But actually it's just there to look cool.

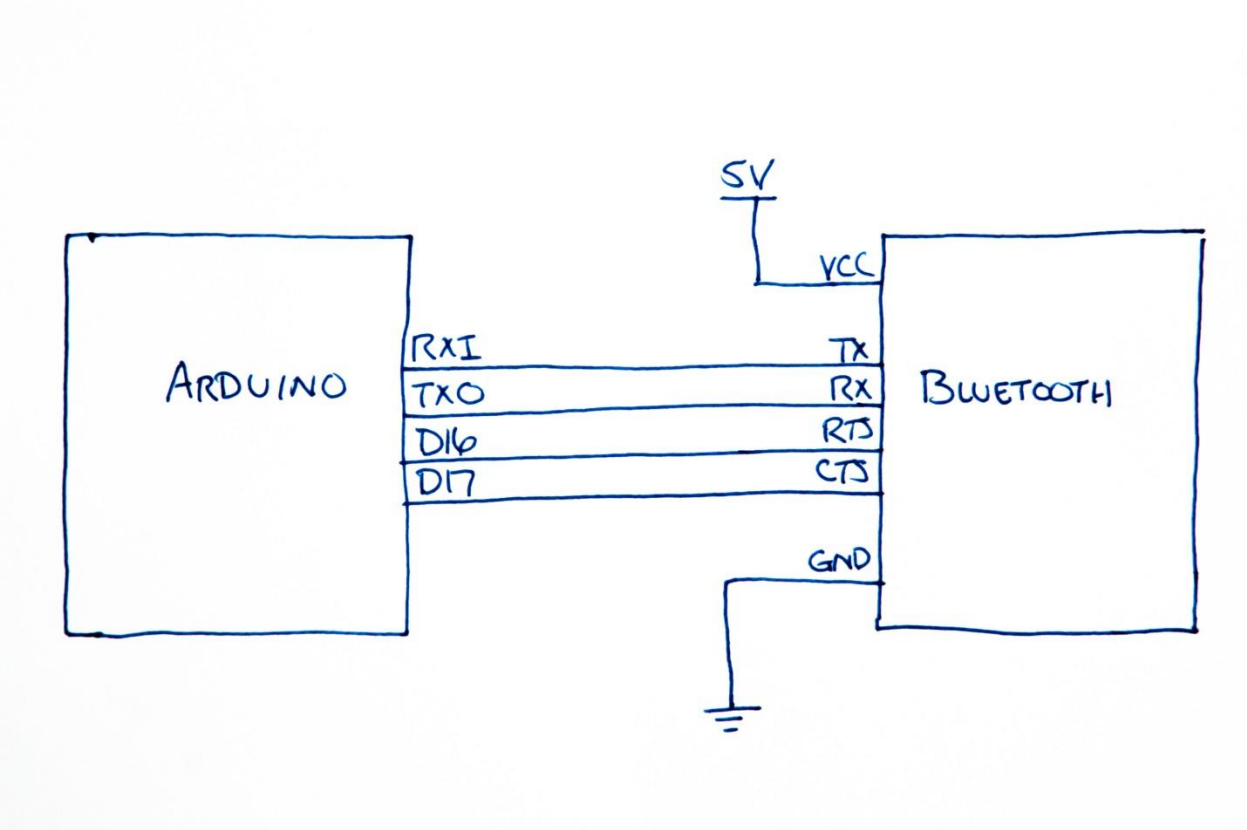
Power Supply and Regulation

Power in the transmitter is provided by three AAA cells. These provide a nominal 4.5 volts that supplies the RF module directly. They also feed a Pololu boost regulator that generates 6 volts for the Arduino. The Arduino's own regulator drops this down to 5V for the ATmega. And the 5V output of the Arduino is used to power the LCDs, joystick and Bluetooth modules.



Bluetooth Module

Did I mention bluetooth? Yes, I stuck this in but haven't used it yet. It could potentially be used to link to a smartphone providing features like a calibration, GPS logging of a KAP session, etc.

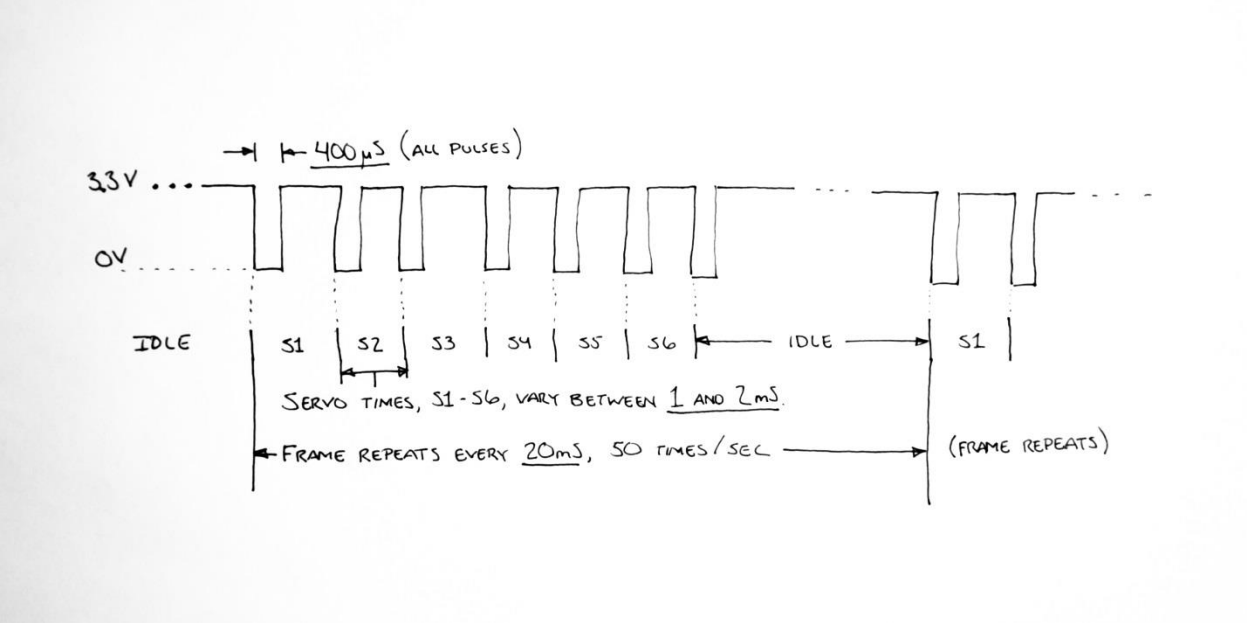


[Correction: the diagram shows D17 where it should say D14. Also, the RTS and CTS pins were not wired up as they are not necessary.]

Appendix A: PPM Signal Generation

The key to interfacing the Arduino with the HK-T6A transmitter module is the PPM Signal.

A PPM signal represents the six servo positions with a series of pulses. The time intervals between pulses correspond to servo positions. The description below refers to this diagram:



For a system like ours with six servos, there will be seven pulses that repeat periodically. The time between pulses gives the angles of the servos. The first servo position corresponds to the time from the first pulse to the second. Similarly the second servo position is indicated by the time from the second pulse to the third. The pulses themselves are always 400 microseconds long but that doesn't matter. It's the time *between* pulses that determines the servo positions.

An interval of 1.5 milliseconds, for example, would put the servo in the middle of its range of motion. If the time interval were shorter, the servo would position itself to the left, longer and it would go right. Typically the time interval varies from 1.0 to 2.0 milliseconds but the Hobby King RF module can support timing from 0.7 to 2.3 milliseconds, giving a little extra range to the servos we connect.

The position of the last servo, number six, is determined by the interval between pulses 6 and 7. After that there should be no pulses for a longer time interval. (About 10ms. It varies). The whole process repeats every 20 milliseconds, or 50 times per second. Each repetition is called a frame. The long idle period between the last pulse of one frame and the first pulse of the next frame is how the receiver knows which pulse is which.

In older R/C systems, the PPM signal was closely related to the modulated RF signal. But 2.4 GHz systems, with their digital radios and spread spectrum schemes, don't actually use the PPM format anymore. Still, the transmitter module in the HK-T6A was apparently designed to replace one of those

older R/C transmitters since it accepts a PPM signal as its input. This is fortunate since it makes interfacing with an Arduino quite easy.

The Arduino we've chosen has special hardware that can produce this PPM signal. To get right down to it, the peripheral called Timer 1 is capable of producing the PPM signal on pin 10. It needs a bit of help from the software, though. The PPM generation process works like this. We set up the timer with two time periods: the pulse width (400uS) and the interval until the next pulse (based on the position we want the servo to have.) The timer hardware sets the output pin low for the pulse width then high for the rest of the interval. At this point, it signals the software it is done, so we load the hardware with values for the second interval. The process repeats for intervals 3, 4, 5 and 6, then we program it for one long interval that is the time until the next frame. The process repeats indefinitely. The software simply has to start the timer running when the system powers on, then repeatedly set the pulse intervals from an interrupt.

You can refer to the ATmega32U4 documentation on Timer 1 for every gory detail.

In terms of software, we need an interrupt service routine to update the timer for each interval and a bit of code to initialize things and get the ball rolling.

Here's the interrupt service routine I use:

```
#define PPM_FRAME_LEN (64000) // 32mS -> 30Hz rate
#define PPM_CHANNELS (6)

struct Ppm_s {
  int phase; // which phase of PPM we are in, 0-6.
  int time[PPM_PHASES];  // width of each PPM phase (1 = 0.5uS)
  bool startCycle;
} ppm;

ISR(TIMER1_OVF_vect)
{
    static unsigned remainder = PPM_FRAME_LEN;

    // Handle Timer 1 overflow: Start of PPM interval
    // Program total length of this phase in 0.5uS units.
    ppm.phase += 1;
    if (ppm.phase > PPM_CHANNELS) {
        ppm.phase = 0;
        ppm.startCycle = true;
    }

    // Remainder computation
    if (ppm.phase == 0) {
        // Store remainder
        ppm.time[ppm.phase] = remainder;
        remainder = PPM_FRAME_LEN;
    }
    else {
        remainder -= ppm.time[ppm.phase];
    }
```

```
        // Program next interval time.
        OCR1A = ppm.time[ppm.phase];
    }
```

And the code to initialize and start the timer:

```
    void ppmSetup()
    {
        pinMode(PPM_OUT, OUTPUT);
        pinMode(TX_PAIR, INPUT);

        // Init PPM phases
        int remainder = PPM_FRAME_LEN;
        for (ppm.phase = 1; ppm.phase <= PPM_CHANNELS; ppm.phase++) {
            ppm.time[ppm.phase] = PPM_CENTER;  // 1.5mS
            remainder -= ppm.time[ppm.phase];
        }
        ppm.time[PPM_RESYNC_PHASE] = remainder;
        ppm.phase = 0;
        ppm.startCycle = true;

      // Disable Interrupts
      cli();

      // Init for PPM Generation

      // Fast PWM with OCR1A defining TOP
      TCCR1A = WGM_15_1A | COM1A_00 | COM1B_11 | COM1C_00;
      // Prescaler : system clock / 8.
      TCCR1B = WGM_15_1B | CS1_DIV8;
      // Not used
      TCCR1C = 0;
      // Length of current phase
      OCR1A = ppm.time[ppm.phase];
      // Width of pulses
      OCR1B = PPM_PULSE_WIDTH;
      // interrupt on overflow (end of phase)
      TIMSK1 = TIMSK1_TOIE;

      // Enable interrupts
      sei();
    }
```

Again, that's more technical information than I can explain here but if you cross reference this code with the Atmel documentation, you should be able to produce a PPM signal with your own system. (Specifically, see Chapter 14. 16-bit Timers/Counters, of the ATmega32U4 datasheet.)

With the timer running and this ISR in place, the rest of the system simply has to write values to ppm.time[N] to set each servo's position. Servo positions are represented by the pulse timing, measured in half-microsecond ticks. So a servo in the center position (1500uS interval) would be represented by the number 3000. Values should stay in the range 1400 to 4600 or the RF transmitter

and receiver will malfunction. ppm.time[0] is the time interval between frames and shouldn't be modified by the main software. Servo 1 is controlled by ppm.time[1], etc.

Each time the PPM frame completes, the interrupt sets the ppm.startCycle flag. The loop() function of the Arduino code looks for this and uses it to time all the polling processes that should run at about 50 Hz.

Appendix B: Arduino Pin Assignments

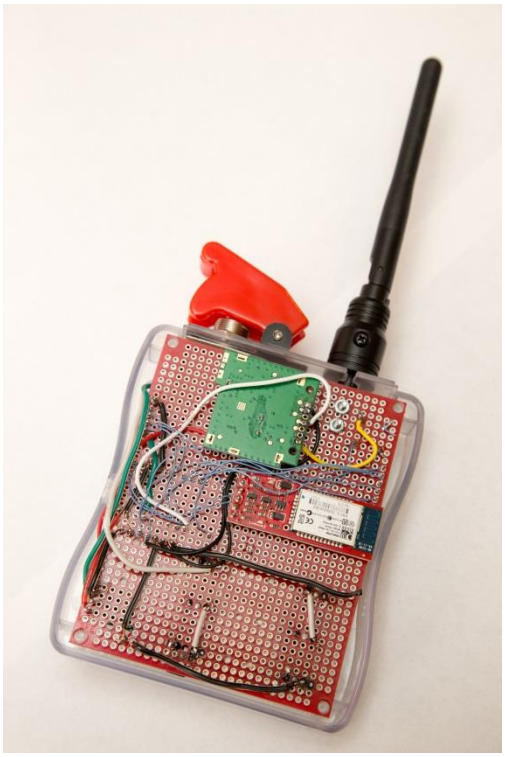
The following table summarizes which pins of the Arduino are used for which functions:

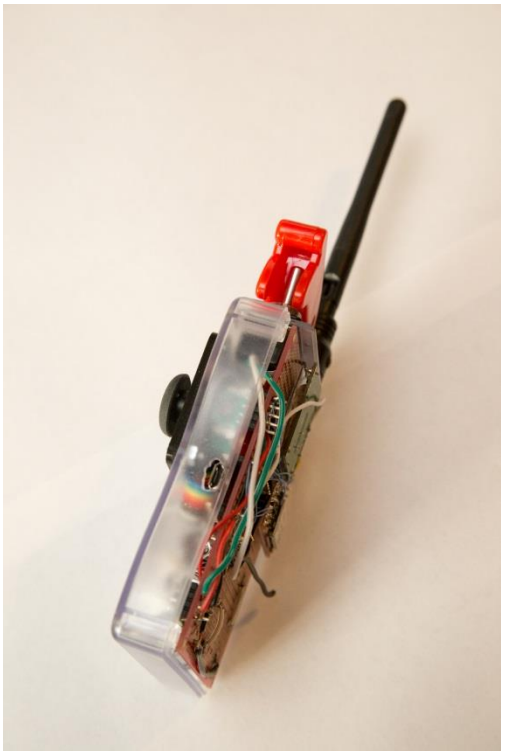
Arduino	ATmega 32U4	Function	Notes
GND		Ground	
RAW		6V	From Pololu voltage booster. Would have been nice to use 4 AAA cells instead of 3 and eliminate this regulator but they wouldn't fit in the case!
VCC		5V	Output to peripherals: Bluetooth, joystick, LCD
RST		Reset	Not connected
RXI (D0)	20 (Rx)	Bluetooth Rx	Not implemented yet.
TXO (D1)	21 (Tx)	Bluetooth Tx	Not implemented yet.
D2	19 (SDA)		Future I2C
D3	18 (SCL)		Future I2C SCL
D4	25 (PD4)	LCD CLK	
D5	31 (PC6)	LCD DI	
D6	27 (PD7)	LCD CS	
D7	1 (PE6)		Future Bluetooth RTS
D8	28 (PB4)		Future Bluetooth CTS
D9	29 (PB5)	Joystick button	Polled but this pin supports interrupts if that becomes desirable.
D10	30 (PB6)	PPM Signal	Pin 10 must be used as it is associated with Timer1.
D14	11 (MISO)		Future SPI Bus, MISO
D15	9 (SCK)		Future SPI Bus, SCK
D16	10 (MOSI)		Future SPI Bus, MOSI
A0	36 (ADC7)	Joystick X	
A1	37 (ADC6)	Joystick Y	
A2	38 (ADC5)		Future
A3	39 (ADC4)		Future

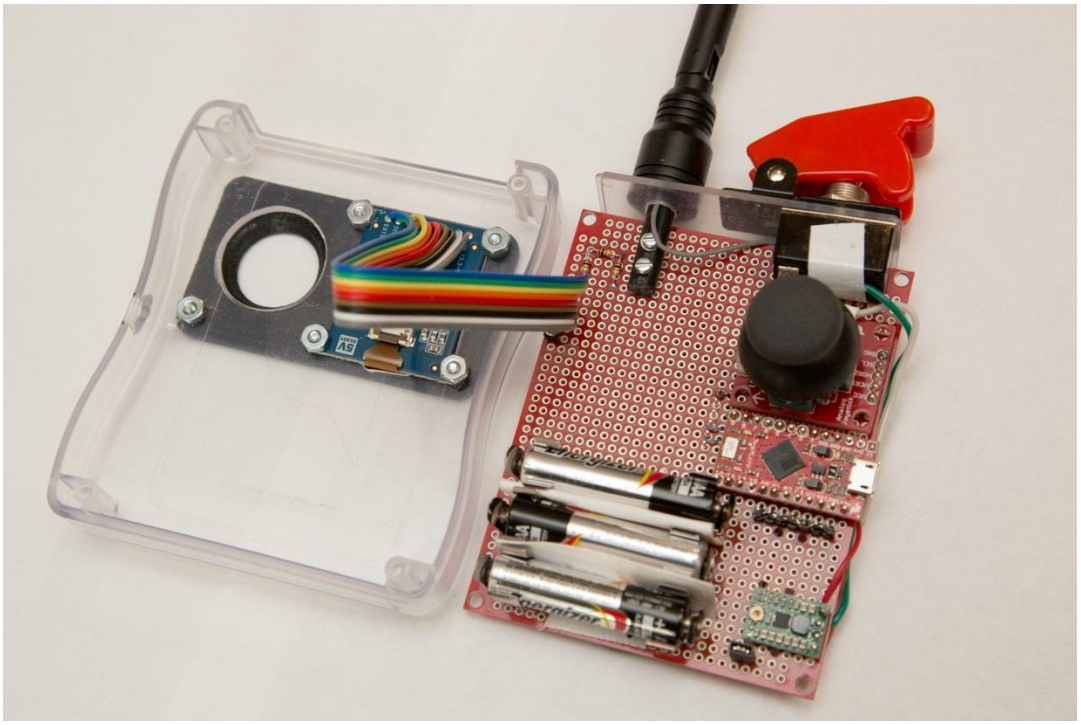
In selecting these pins, I tried to keep some useful functions available even though not used in this design. The Arduino I2C bus pins are open as are the main SPI pins and 4 general purpose I/O lines (two of which support analog input). These create the potential to extend the design in a number of interesting ways.

Appendix C: Assembly and Packaging

Here are a few photos showing how the hand-soldered board fits together and mounts in the case:







Note: the photos show the bind signal (white wire from RF module) connected to D15 of the Arduino. That connection should be omitted. See discussion of bind signal with RF module, above.

References

This project was inspired and informed by several other sources. The interested reader may want to refer to these in their own KAP explorations.

Cris Benton's repackaged R/C

http://arch.ced.berkeley.edu/kap/wind/?p=35

David Mitchell's repackaged R/C controllers

https://www.flickr.com/photos/dave_mitchell/10514492995

https://www.flickr.com/photos/dave_mitchell/3551746790

Scott Haefner's repackaged R/C

http://scotthaefner.com/kap/equipment/controller

PPM Signal Reference

http://sourceforge.net/p/arduinorclib/wiki/PPM%20Signal/

ATmega32U4 Documentation

http://www.atmel.com/devices/atmega32u4.aspx?tab=documents

Adafruit GFX Library

https://learn.adafruit.com/downloads/pdf/adafruit-gfx-graphics-library.pdf

Adafruit SHARP Memory Display instructions

https://learn.adafruit.com/adafruit-sharp-memory-display-breakout/programming