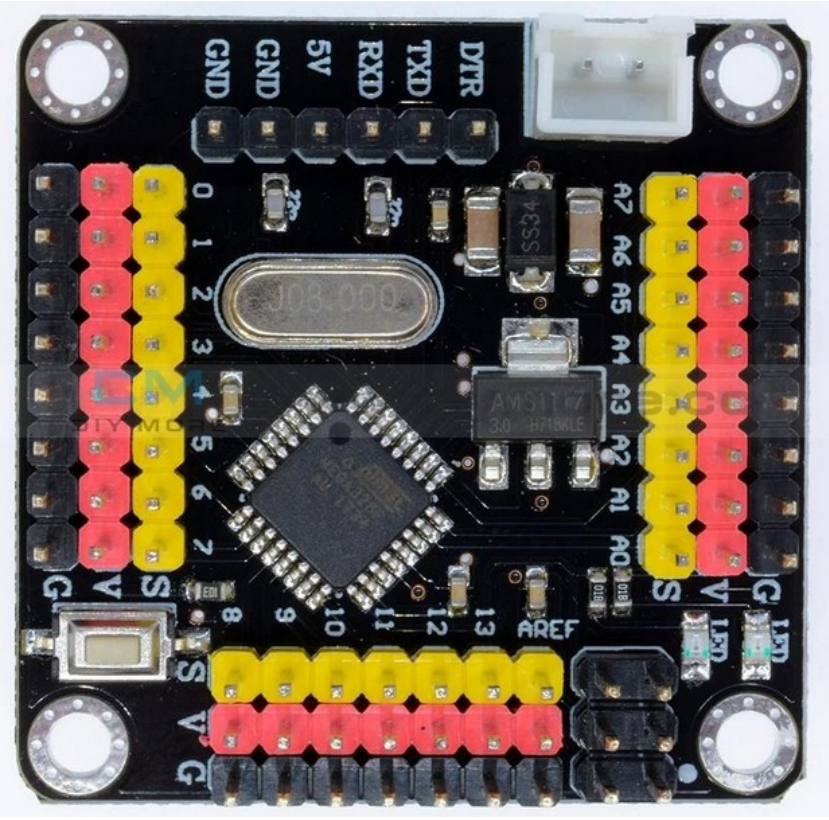
FrSkyV8Tx library

Make your own R/C transmitter using an AtMega328 Arduino, and a CC2500 module. The electronic parts only cost a handful of pounds (or dollars). The transmitter can be used with FrSky V8 and compatible receivers. The same transmitter hardware is also compatible with other FrSky compatible receivers (D8 and D16 modes), plus Corona, and Futaba SFHSS.

Important: the CC2500 modules operate at 3.3V. This includes the supply (VCC) AND the logic signals (MOSI, MISO, SCK, CS). You must use a 3.3V Arduino, or a level shifter for the logic signals if you're using a 5V one.

Arduino options:

1. (Preferred) use a 3.3V Pro Mini or Pro Mini Strong. Then you can use the on-board 3.3V regulator to power everything (including the small or large CC2500 modules). You can power the Arduino via a battery connected to its Vin (Raw) pin. Any battery voltage from about 5V to 12V is fine. You could use four AA pen-cells or two 18650s or a two-cell LiPo etc. AtMega328 chips are supposed to run at less than 16MHz when powered by 3.3V so most such boards will have 8MHz crystals. This is fine as 8MHz is plenty fast enough for an R/C transmitter.



2. Use a 5V Arduino, but power it from your own regulated 3.3V supply into its VCC pin (not Vin/Raw). You can use a 16 MHz Arduino – yes it will almost certainly work okay at 3.3V.

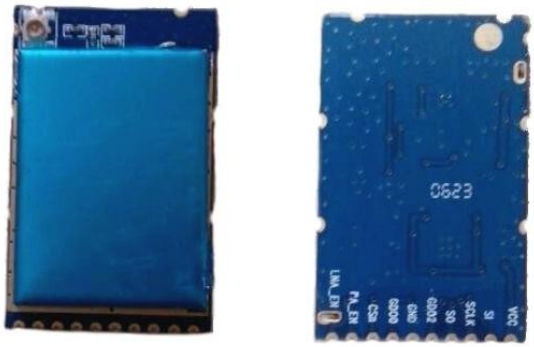
3. Use a 5V Arduino, running at 5V, but use a logic level shifter on the digital connections between the Arduino and the CC2500. If your 5V Arduino has a 3.3V pin, you can use that to power the smaller, limited range CC2500 modules, but the 3.3V supply on 5V Arduinos is

usually incapable of powering the larger CC2500 modules that have higher output power – so you will need to provide your own 3.3V regulator to power the CC2500 module – and supply that regulator direct from your battery rather than through the 5V Arduino.

CC2500 options:

The small modules, featuring just the CC2500 chip itself with a few tiny passive components, and an on-board PCB antenna, are low power, 'park-fly range.' I've tested them out to about 200 metres ground range, so they're fine for cars, boats, indoor aircraft, or lighter, slower, outdoor "park-fly" types. I wouldn't recommend them for fast, heavy, or expensive flying models.

The one pictured above the connections are (left to right): 1:GND 2:(3.3V) 3: SI (MOSI) 4:SCK 5:SO (MISO) 6:no connection 7:no connection 8:CS (most likely Arduino pin 10, but see below).



The larger modules, example pictured above, can connect to a 'proper' antenna (often not supplied with the module, but you can buy the antennas separately, or salvage one from an old R/C transmitter or a WiFi router). Don't power up the larger modules unless an antenna is connected. These modules, equipped with a good antenna, have a transmitting range equaling or surpassing that of most commercial R/C transmitters.

The connections:

The default connections between the Arduino and the CC2500 module are:

Arduino	CC2500
10	CS or RFCS or CSn
11 (MOSI)	SI (MOSI)
12 (MISO)	SO (MISO)
13 (SCK)	RFSCL or SCLK
VCC (only if Arduino is 3.3V type – see above)	3.3V
GND	GND
No connection	GD02 linked to LNA_EN (if present)
No connection	GD00 linked to PA_EN (if present)

The library:

I wrote the library to provide a simple way to use the CC2500 for Arduino-based R/C transmitter projects. I studied the wonderful **DIY-Multiprotocol-TX-Module** by Pascal Langer and others, to work out how to do it.

DIY-Multiprotocol-TX-Module is great, but it's large and complex so it puts most Arduino coders off. If you want to use it directly to write your own transmitter code, you have to tightly integrate the DIY-Multiprotocol code into your own (the timing of the function calls is critical).

In contrast, FrSkyV8Tx is simple to use. All the complexity is hidden away, so you just need to call start() to start it transmitting, and setChannel() whenever you want to update one of the transmitted channel positions. There are no timing requirements, because all the communication between the Arduino and the CC2500 modules is handled by (hidden in the library) timer interrupts.

Installing the library:

Look in your Arduino folder (where your sketches are) and you'll see a libraries folder. Put the FrSkyV8Tx folder inside your libraries folder. Then restart your Arduino IDE.

Trying out the library:

Inside your Arduino IDE, go to the File menu, Examples, and scroll down till you find FrSkyV8Tx. Choose the bareMinimum sketch. You'll see that it contains about five lines of actual code, plus some comments. You should be able to upload that sketch to your Arduino and get it to bind with your FrSky V8 receiver. The bareMinimum sketch runs in bind mode for ten seconds each time it is restarted, and then swaps over to normal transmitting. See the comments in the sketch about experimenting with the FINE_TUNING value if you can't get it to bind with your receiver.

The other library examples:

At the time of writing, there are two other 'example' sketches for the FrSkyV8Tx library, besides the bareMinimum one described above:

- basicFourFunction This is really just for study, not for controlling a real model! It doesn't implement things like trim buttons, rate switches, etc. But it's just a few lines of code, and

shows how you can make a working transmitter by connecting the four joystick pots of a transmitter up to the Arduino.

- cppm This is a fairly complete example you could use to convert an existing transmitter that has a cppm output signal. Transmitters with a buddy-box (trainer) connector provide this signal on that connector, and older transmitters that don't have a buddy-box connector usually still have a cppm signal available internally. The example shows how you can implement auto-binding (like the bareMinimum sketch) or have a bind button. Also how you can implement a trim pot connected to an analog input for fine tuning, and how you can implement a 'range test button'.

The API of the Library:

API stands for Application Programming Interface – the functions ("methods") available in the library.

#include <FrSkyV8Tx.h>

This includes the library in your sketch. The Arduino IDE knows to look in your Arduino/libraries folder to find it.

FrSkyV8Tx frsky(TX_ID, CS_PIN, FINE_TUNING);

This uses the "class constructor" to "create an instance of the class" You put this up near the top of your sketch – before the setup() or loop() functions. You don't have to use #defines to name the three parameters, but it's good practice. It would work just the same if you put in the parentheses (12345, 10, 0); - this would set the transmitter ID to 12345, us pin 10 as the chip select line, and use a fine tuning value of zero.
Calling the constructor like this gives the instance of the class a name 'frsky' in this example, but you can use whatever name you want, and tells the class three numbers that it needs to know to enable the transmitter to work:

- The transmitter ID. This is the unique number that the transmitter sends all the time when it's transmitting. This is the number your receiver remembers when it's bound to a transmitter, and from then on the receiver ignores any received packets of data that don't include this number.
- Which Arduino pin to use to drive the CC2500 Csn (chip select pin). You would normally choose pin 10 for this (because you'll already probably be connecting pins 11, 12, and 13 to the module for MOSI, MISO, and SCK) – but you can choose any pin you like, including the analog pins A0 to A5, if you wish.
- The fine tuning value to use. This is described in the 'Important notes;' section below.

FrSkyV8Tx frsky(TxID, CS, MOSI, MISO, SCK, FINE_TUNING);

This is an alternative form of the constructor – use one or the other but not both. This allows you to choose different pins from 11, 12, 13 for MOSI, MISO, and SCK. For example you could put FrSkyV8Tx frsky(12345, 7, 5, 6, 4, 0); to use pin 7 as the chip select, 5 for MOSI, 6 for MOSI, and 4 for SCK. In this example, the TxID would still be 12345 and the fine tuning value zero. These pins are the ones used in the early commercially available multiprotocol modules that were AtMega328 based (the newer modules use an STM32 chip). Unless you have a good reason to use this 'bit banged' SPI protocol method (for example you have one of the old modules and want to use it with this library), then you're advised to use the first form of the constructor that uses the hardware SPI pins, 11, 12, 13. Hardware SPI takes up less microprocessor time.

frsky.setTransmitterID(TxID);

This allows you to change the transmitter ID after you have used the constructor to create the class instance. Normally there's no need to do this, but if you want to implement a "model match" system in your transmitter, so that it uses different IDs for different model memories, this could be useful – without this you'd need to switch the transmitter off and back on when changing model memories (and therefore transmitter ID). The new TxID is used immediately – if the transmitter is already transmitting it will continue to do so, but with the new ID. If it's not currently transmitting, then it will use the new ID the next time you call the .start() method.

frsky.setFineTune(fineTuneValue);

This allows you to change the fine tuning of the transmitter after you used the constructor. Useful if you want to implement fine tuning from a trim pot, or from a menu when your transmitter has a display screen. The new fine tune value is used immediately – if the transmitter is already transmitting it will continue to do so, but with the new tuning. If it's not currently transmitting, then it will use the new fine tuning value the next time you call the .start() method.

frsky.setTransmitterPower(powerValue);

This can be called when the module is already transmitting, or when it's stopped. It sets the transmitting power to the number specified: this ranges from about 80 for very low power up to 255 for full power. If you don't use this method, then the library automatically uses full power in normal mode. The library automatically uses low power (80) when binding – no matter what value has been set using this method. After binding it returns to using the power specified by this method (or the full power default if this method has not been called). Commercial transmitters and modules set a value of 80 when you press the range test button or select range test from a menu. You may wish to copy this value if you want to implement your own range test feature. See the cppm example sketch to see how this can be done. Power values lower than 80 are not often used – and if you go much below 80 the transmitter will no longer transmit a signal powerful enough to be received – even when the receiver is very close by.

frsky.start();

This starts the module transmitting. If it's already transmitting then this call is ignored. If .bind() has been called while the module is stopped, then when it is started it will be in bind mode for the first ten seconds, after which it will switch to normal transmission mode.

frsky.stop();

This stops transmission immediately, whether in normal transmitting mode or bind mode. If .bind() was previously called while the module was stopped, then calling .stop() cancels the auto-bind that would otherwise occur on the next .start() call.

frsky.bind();

This enables a ten-second period, during which time the transmitter sends a bind signal rather than its normal control signal. If the transmitter is already working (started) then it will immediately enter bind, and after ten seconds of binding will return to normal transmitting. If the transmitter isn't working (not started) when bind is called, then the bind mode will be entered for the first ten seconds the next time that .start is called.

frsky.setchannel(channelNo, microseconds);

This is the most used method. Your sketch will probably need to call this hundreds of times per second to update the data sent to the receiver - and hence the positions that the servos will be driven to. You don't need to update a channel unless its position has changed, but you can if you wish.

channelNo runs from 0 to 7 for the eight possible channels, microseconds sets the position for the specified channel number. The centre position is 1500 microseconds, and the standard normal range is +/- 512 from there – so 988 to 2021. You can use an extended range of up to 150% of the standard range, so the microseconds value can range from 732 up to 2268. Values outside that range will be clipped to those limits. Channel numbers greater than 7 will be ignored.
The FrSky V8 protocol transmits packets of data every nine milliseconds, and basically alternates between sending the first four channels and the last four. Actually there are five packets in a complete cycle, and the first four channels are sent three times, and the last four twice. So regardless of how often you call setChannel, there can be a delay (latency) of up to nine milliseconds before the first four channels are actually updated at the receiver – and in the worst case, up to eighteen milliseconds delay for the last four channels.

frsky.setDeglitchedChannel(channelNo, microseconds);

This does exactly the same as setChannel, but attempts to remove any spikes of noise that may be present in the microseconds values you pass. It works by comparing the latest value you pass with the previous two values. It selects two values from those three which are closest to each other, and then uses the most recent of that pair. The result is that if you pass values like 1505, 1508, 1503, 2100, 1507, 1511, … then the 2100 value will not be used. This can be useful if your transmitter hardware is subject to noise – maybe from rough stick potentiometers, or electrical pickup from a buzzer or similar – it can help that noise from being reproduced by glitching servos. But there is a price to pay of increased latency: so if you only call setDeglitchedChannel twenty times per second, then when a genuinely steady signal begins to genuinely change, there will be an extra delay of one twentieth of a second before that movement is transmitted to the servos. I recommend using the normal setChannel method as a default, and only swap to using setDeglitchedChannel if you're experiencing servo glitches that you can't eliminate by improving the hardware.

frsky.micros();

This works exactly the same as the normal Arduino micros() function, except that it's more accurate. Both functions return the number of microseconds since the Arduino was switched on, in a 32-bit number from 0 to just under 4.3 billion. That number of microseconds gives about 71.6 minutes, so if you ever leave your Arduino switched on for that long, the number wraps around from the very large number back to zero, and starts over. Normally we're only interested in the difference between two such timestamps, and if you do all the calculations using 32-bit unsigned integers, then no error will be introduced in code like this:

```
uint32_t startMicros = frsky.micros();
// do something that takes some time
uint32_t endMicros = frsky.micros();
Serial.print("That operation took ");
Serial.print(endMicros – startMicros);
Serial.print(" microseconds\n");
```

Even when the wrap-around occurs between the two timestamps, the result of subtracting the first timestamp from the last on will give the correct (positive) number of microseconds.

The normal Arduino micros() function only returns values that go up in steps of four (on a 16 MHz Arduino) or eight (on an 8 MHz Arduino). The frsky.micros() values resolve to single microseconds on both types.

Important notes:

Don’t leave the TX_ID, defined up near the top of the sketches, at the default 12345. Each transmitter (that’s switched on at the same time as other FrSky V8 transmitters) must use its own TX_ID. You can choose any number less than 32768, so maybe some digits from your phone number or something else that will likely be unique to you. If you’re building several transmitters, and you know that you’ll only be using one at a time, then you can use the same TX_ID on all of them. It’s the TX_ID that receivers bind to, during the bind process, so if you use the same TX_ID on all your transmitters, then you can control any model from any transmitter without rebinding. This is potentially dangerous, if you have different control arrangements or trims for different models – so you may prefer to use unique TX_IDs for each of your transmitters.

The “fine tuning” thing is something that all CC2500 chips need. Once you’ve established the correct fine tuning value, then you can use that same value for ever. If you control different models with the same transmitter, then you don’t have to change the value when you change models.

With commercial sets or modules from FrSky, you won’t have seen “fine tuning” before, because FrSky factory calibrate all their transmitters and receivers to work at the optimum frequency. The tuning value is then stored in an EEPROM, so even if you upgrade the firmware of a transmitter or receiver, the new firmware continues to read, and use, the same stored fine tuning value.

However if you’ve used one of the commercial multiprotocol modules, then you’re probably already familiar with fine tuning (if not, then you should be!) Whether you’re setting fine tuning from the menu of a modern transmitter, or by twiddling a tuning pot, or by trial-and-error experimentation with your Arduino sketch and the FrSkyV8Tx library, the procedure is the same: Adjust the value till your receiver starts to flash its LED indicating signal lost. Note the minimum and maximum values at which this occurs, then set the number (or pot position) half way between those two values. Once you’ve set it you’re done, and should never need to change it again unless you swap to a different CC2500 transmitter module.

See the thread on the Mode Zero forum about this project:
http://mode-zero.uk/viewtopic.php?f=42&t=1092