S/C Transmitter with LCD Display and 27 Model Memories

There's an old guy in our (electric only) club who used to fly single channel and reeds. He has built a model called a 'Gasser' that he flew single channel back then. He was flying it with modern (Futaba FF9 with Spektrum module) gear - but the linkages to the rudder and elevator are done with torque rods from the servos, so the back end of the plane looks like a genuine escapement version. He was trying to fly it 'bang bang' by only using either full stick movement or leaving the sticks centred - but I told him I could do better than that and build him a real push button set that would accept his Spektrum module.

So I coded up an Arduino to mimic one of the more complex set-ups with rudder, kick up elevator and quick blip to toggle the motor on and off - but I figured we needed menus to set-up the servo centre positions and amount of throws, so I added an LCD screen with a small joystick to navigate the menus and edit the values. And I added some other modern conveniences like multiple model memories, a countdown timer with a bleep countdown for the motor stopping, motor soft start and so on.

We've flown his Gasser using my 'single channel' transmitter as a buddy box slave into his transmitter, so he could take over and save the model if something went wrong. It was fairly successful but he needs to make some adjustments to the motor thrust line and move the centre of gravity before the next test.

So, as I was getting impatient, I dragged my old Veron Cardinal (diesel powered, free flight) down from the loft and converted it to radio control electric power. I fitted some Corona gear to it as the only futaba-shaped module I have is an old Corona one. (I have lots of JR/Turnigy/Taranis sized modules, so the next single channel I build will be to fit those).

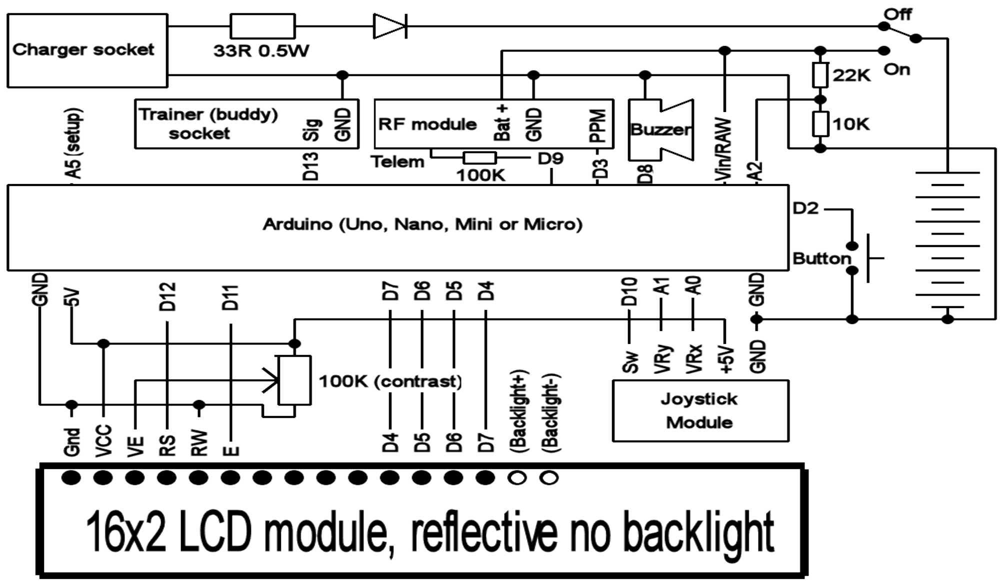
Anyway, I test flew the Cardinal last Friday morning, early when it was calm, and it worked great! I messed up the controlling on my first ever go at single channel and landed in a cornfield, but with no damage. Then I had about six more flights and landed on our (small) runway one time but got fairly close by on the other attempts. Great fun.

As you can see, it's a 3D printed case and it uses an Arduino Pro Mini to do most everything. I thought about fitting a LiPo or LiFe battery, but worried about how my old pal would charge it and balance the cells - so I ended up with a 6-cell NiMH pack. I fitted a dropper resistor and protection diode so he can charge it with one of his existing Futaba chargers, into a normal Futaba transmitter charging socket.

It works on any 16MHz AT328 Arduino (Uno, mini, micro, nano, etc.) It could be adapted to run on a 12MHz version or other processors but it uses one of the AT328 chip's hardware timers to divide down the 16MHz clock and drive out the PPM signal under interrupt control. The signals have a resolution of half a microsecond, but I didn't need that accuracy in the single channel 'menus', so you set the centre positions, and travel limits by specifying the pulse spacing in microseconds (these are the numbers you would read off one of those modern servo testers with a digital readout: the range is roughly 800 to 2200 with the centre position being 1500). Like I say the PPM is all output under interrupt control, so from the main sketch, you just do:
cppm_out::start();
from your setup() method, and then to alter a channel's setting at any time, you do, for example:
cppm_out::set_uS(2, 1000);
that example would set channel 3 (there's a 2 in the example as the channel numbers begin at 0 in the code) to 1000 uS, so if channel 3 was the motor speed controller, this would switch the motor off.

I've also got an interrupt driven button monitor routine that captures the times when the button is pressed or released. The main loop() function then looks at these times, does the necessary switch contact de-bouncing and then decodes the button presses to work out what to output to the ppm driver. The main loop also checks the analogue inputs from the small joystick that allows the menus to be navigated and it runs the timer (motor run time) and buzzer and so on.

Here's the circuit schematic.



Oh, someone has pointed out that the code in the .zip file wouldn't compile if you put it in a folder named, "singleChannelTxSketch" It was my fault for renaming the .zip putting 'Sketch' on the end before I uploaded it. I've now removed that .zip archive and uploaded a new one with the correct name. Make sure the folder where you unzip the files to is named singleChannelTx (this is the same name that the .ino file has). It should compile okay then if you have a new enough version of the Arduino IDE. I'm using 1.6.5 but for all I know you'll get away with using an older version, providing it's not too ancient.
It probably looks like a mistake that I've got D4, D5, D6 and D7 going to the same-named pins on the LCD but in the reverse order. The diagram is right though, it's just a coincidence that it ended up that way. The pins on the LCD module aren't actually labelled (at least mine weren't) but I've drawn them out in the correct order to match an actual LCD, looking at the front (display) side. The four unused LCD pins are named D0, D1, D2 and D3 in the LCD documentation.
You can see that, on the prototype, I did use a backlit LCD. This is just because I had one lying around in my bits and pieces box (same for most of the other components actually). But when I build another one, I shall use a non-backlit one: the backlit one is pretty good in strong sunlight, but like most backlit displays it's not ideal and I think a regular non-backlit LCD will be better for outdoor use. The backlit ones are actually cheaper on Ebay than the reflective ones - you can get a backlit one for about £2, but the reflective ones cost £3.50 or so - and about £5 including postage if you only buy one. If you use a backlit one, then you have to connect the backlight or you can't see anything, even in daylight! It's okay just to connect straight to the Arduino 5V and GND lines - the current draw is well within the capability of the Arduino regulator, and would take all day, or longer, to run your battery down. The backlit LCD modules have a built-in voltage dropping resistor, so there's no need to use an external one.

You can see the module plugs into the back just like on a real Futaba transmitter! I don't actually own a Futaba transmitter that accepts modules anymore, so any future development of the case may go in the direction of JR-sized modules (these are the ones that Turnigy and FrSky transmitters also use).

The hole at the top is the 'sound hole' for the buzzer. I used an old JR receiver on/off switch as it was lying around, but obviously you could use any on/off switch that fits. Then there is a Futaba-style charging socket (JR/Spektrum use the same connector but with opposite polarity!) You can use any NiCad/NiMH Futaba charger with this one - the very old ones charge an 8-cell NiCad pack at 50mA, the newer ones charged at 100mA. Either is fine for this 6-cell pack (there is a resistor in the charging circuit that drops the charge current back to about what it was - even though the transmitter is only using a 6-cell pack. With modern batteries you can leave it on charge forever if you like, but an overnight charge will give plenty of life for a day's flying. The last connector down is a JR/Spektrum compatible buddy box port. It also works with Futaba providing you have the right lead that goes from a 3.5mm mono jack plug on the one end to the Futaba square plug thing used on the latest Futaba transmitters.

Forgive me! It's only a prototype and started out as a mess of components on breadboard and loose on the table before I shoved it in the box! The joystick and LCD and connectors are properly mounted in the 3D print design, but the Arduino and the batteries I just bodged in with hot glue! The Arduino Pro Mini doesn't have any mounting holes anyway, so that's my excuse! The vero board down at the bottom holds the pot for the LCD contrast, the resistor divider for the battery voltage so that the Arduino can monitor the battery, the charger resistor and diode, and provides somewhere to connect all the GND and 5V wires together.

These are the bits you need if you want to make one. (I just realized I left the buzzer out of the photo, so I'll fix that later). I bought the push button from Maplins, but all the other bits from Ebay. I think the joystick is less than £2, the Arduinos are about £3, and the connectors are in the 'several for a pound' range. The buzzer is a 5V piezo buzzer, also from Ebay. I think I bought five for about £1.50 It's not in the 'components' photo but its just a black cylinder, 10mm diameter and 10mm long with a + and - wire on the back end. They come with a paper sticker over the sound end that says, 'HXD remove seal after washing' - you just peel that off. The spec says it works on anything from 1.5V to 5V, drawing about 6mA, so it's okay to work direct off an Arduino output.

The way the display works is as follows:

First, let's assume that when you switch on, the correct model is already selected (it remembers which one you selected last time, even when it's off).

The top line of the display will be showing "Motor time: 5:00" (assuming you have it set for five minutes - again it remembers the previous setting even when it's switched off).

...and the bottom line cycles through three different pieces of information - each one is displayed for about a second and a half:
" 2 Gasser " this shows the model memory selected (Number 2 here - there are 15 available) and the model name, which can be up to 13 characters long.
"Press joy starts" this is an instruction to remind you how to start the motor - you press the joystick in - as well as moving up/down and left right, you can also push it like a button.
"Battery 7.9V" this shows the current state of the battery. If the battery is low, it changes to, say, "Battery low 6.5V" and the buzzer beeps every second.

Now before you start the motor, you can move the joystick to the right to increase the motor run time in 10-second increments, or to the left to decrease it by the same.

When you press the joystick in, the motor runs up to speed and the top line changes to showing "Remaining 4:59" or whatever, counting down the seconds. It gives you a warning beep at 30 seconds remaining, then another at 10 seconds, and a 5-bleep countdown and long final bleep (like the time pips on the radio) for the last 5 seconds before stopping the motor. The "Press joy starts" thing on the bottom line changes to "Press joy stops", of course.

At any time when the motor is running, you can press the joystick again to stop the motor and return to the initial 'stopped' state. You can use this as a kind of 'emergency stop', as you have the normal facility to toggle motor between paused and running with a 'quick blip' on the main button.
Whether the motor is running or not, the main button controls the rudder and elevator. Press and hold for right. Press, release, press and hold for left. Press twice then press and hold for kick up elevator.
If the motor has been started by using the joystick "button" then it can be (optionally, see later) toggled between running and paused by a quick blip on the main button. When the motor has been paused this way, the "Remaining" display changes to "Paused" and the timer stops counting down until you start the motor again.

The up/down direction on the joystick lets you scroll through all the menus to adjust parameters, but I'll leave that for another post as this one is long enough already.

Okay - on to the menu items.

If you move the joystick down (and release) once you get to the Model selection menu. Then down again for the next menu and so on. You can also scroll up through the menus by using an 'up' movement and release.

I'll list all the menus first, and then describe what they do.

Main display, as described in previous post.
Model selection.
Editing model name.
Copying one model memory to another.
Rudder centre position.
Rudder right position (travel).
Rudder left position.
Elevator centre position.
Elevator up position.
Elevator down position! (see below)
Motor off setting.
Motor on setting (allows motor to run at desired speed - say 55% or whatever).
Motor soft start time (lets motor run up to speed gradually instead of starting with a jerk).
Button speed (allows you to set the 'escapement speed' to suit your personal preference).
Motor control by quick blip? (you can set this to 'no' if you don't want to use quick-blip motor control).
Joystick mode! (see below)
Low battery alarm limit.

In the model select menu, you can see all the available models by using left/right on the joystick. When the one you want is displayed, press the joystick in (its button mode) to select that model.

In the edit name menu, pressing the joystick in allows editing to begin, then left/right on the joystick selects a character position within the name and up/down changes the character. When you've finished editing the name you press the joystick in again. With all these joystick functions, there is an auto-repeat that gradually speeds up if you hold the joystick over for a long time - so you could do three quick up presses and releases to change a G to a J, but if you wanted to change it to a Z, you'd probably hold the joystick up and watch it change rapidly till you got close.

The model copying menu uses left/right to select a destination model memory where you want to copy to. Pressing the joystick in copies the current settings to that model memory, and selects the new copy as the current model.

The 'position' menus all display the current microsecond pulse interval for that channel. In real life you don't take much notice of the numbers displayed - say you're setting the rudder right position: then you would press and hold the main control button to move the rudder on the model; then use the joystick to scroll to the 'rudder right menu and then left and right on the joystick till the rudder is in the position you want. When you're setting the centre positions of the rudder or elevator there is no need to use the main control button, of course. With the main control button released you can go to the 'centre' menu items and then use left and right on the joystick and watch the model controls move in real time.

Joystick mode is a little cheat I built in that might be useful for test flying a model that's not yet been trimmed. You set what looks like sensible positions for left/right rudder and up/down elevator. Then enable joystick mode by scrolling to the joystick menu and pressing the joystick in. Then you can fly the model using the joystick. You have proportional control - like with the right stick on a mode 2 proportional set. The main button toggles the motor between the 'off' and 'on' speeds when you're in this mode. It's rather a cheat and only really intended for trimming flights - but it does give you a way to control the model if the centre positions you guessed were wrong, or if you need to use down elevator. It's no substitute for a real proportional set if that's what you want - not least because the cheap joystick isn't particularly linear and has a pretty big 'dead' zone around the centre stick position. Once you've selected joystick mode, then it remains in force till you switch the transmitter off and back on. Originally I had it so that pressing the joystick in again would switch it back off - but I realized that when you're flying this way you're quite likely to press the joystick by accident which would most likely lead to a crash or fly-away unless you quickly realized what had happened.

If you build one, then, you will want to reset all the model memories to defaults and calibrate the joystick. Power it up and adjust the contrast pot till you can see the display - but the display may be scrolling through all the menus by itself - this is because the joystick hasn't been calibrated yet and the Arduino can't recognize that the joystick is in its centre position.

So power off, link the '9' pin to 'GND' and power back on. The display shows: "Cycle joystick" "Center. Button".

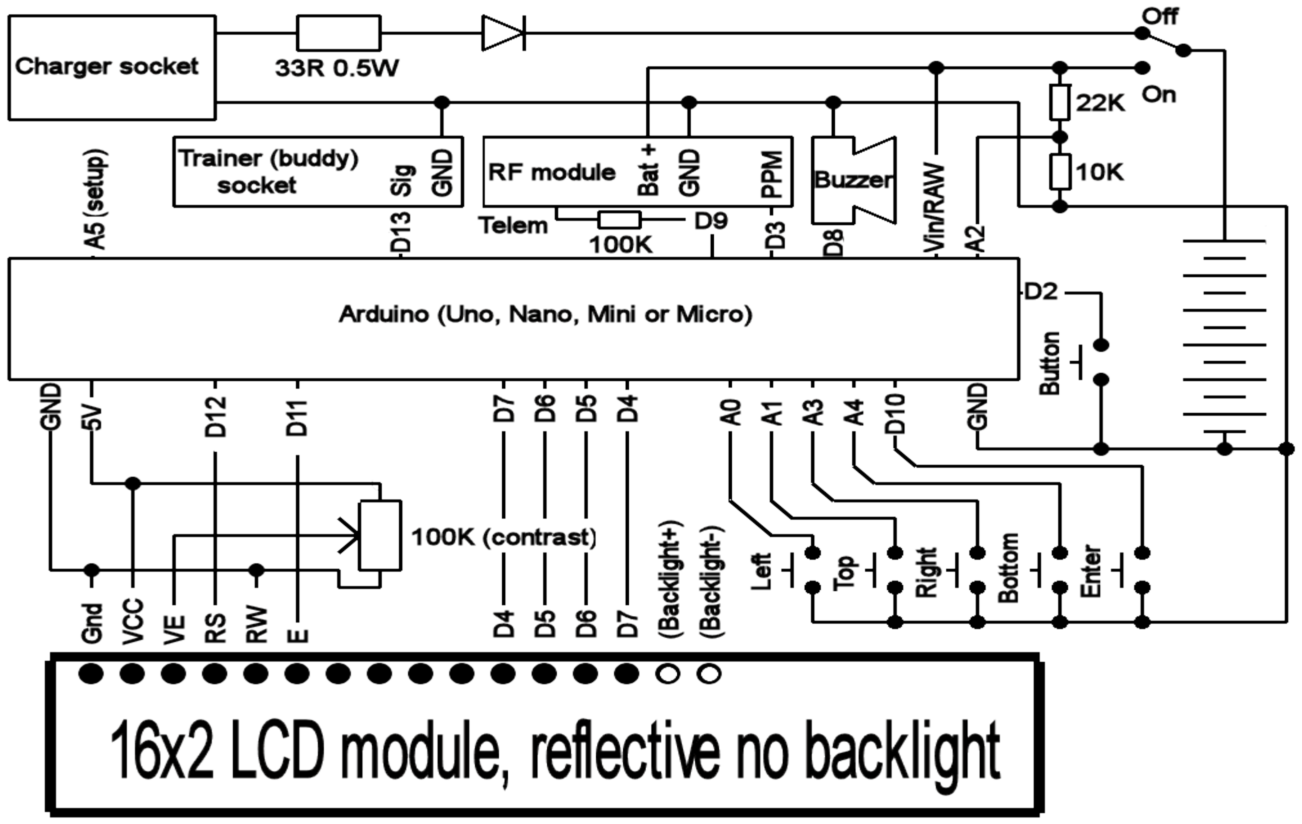
Move the joystick in slow circles at its full travel so the Arduino can read and store the maximum and minimum x and y values. Then release the joystick back to the centre and press the main control button (not the joystick itself). The Arduino then samples and stores the joystick centre position. Now you can power down and remove the '9' to 'GND' link. You shouldn't need to do this again. The joystick settings and the model memories are stored in the EEPROM memory of the Arduino, so even when you modify and upload the program (sketch) the stored values are not affected or lost.

Stay tuned for the 3D printed case info! I'd be happy to print one and send you if you wish. The joystick module is a little tricky to mount - in the 3D print version there is a concave depression inside the front face of the case to match the 'ball' on the joystick, and the mounting legs hold it just the right distance back from the front face.

If you make a case yourself with a hole to suit the joystick and some pillars to mount the joystick PCB, be warned that the pillars tend to catch against the moving part of the joystick mechanism. I ended up with 'leaning pillars' on the 3D print version! If you're fabricating something yourself it's probably easier to use a spreader plate around the joystick PCB so that the pillars can be further away. Alternatively you could mount the joystick from the back and then just have the front cover fit over it without needing any 'front pillars'.

Another way to do it would be to connect the five buttons directly to 10, A0, A1, A3 and A4 (A2 is used to monitor the battery voltage but at present A3 and upwards are unused). That would require a small change to the code but make the circuit simpler. Let me know if you want to do that. I can try to integrate it into the code somehow so that the same version of the code works with either an analogue joystick with a push button facility built-in or with five ordinary push buttons. The analogue inputs on the Arduino work perfectly well as digital inputs too. '0' and '1' digital input are also currently unused, but it's best not to use those as it leaves the Serial port free for debugging. On some of the Arduinos A6 and A7 are also available, should we need them!

Here's the schematic for the five button version (six buttons if you count the main 'single channel' button).



There are pros and cons for this version. Pros are that an analogue joystick seems wrong on a single channel set, and as Ron said, it's too easy to cheat using the 'joystick mode'; also it's easier to mount five push buttons as they just require holes in the front panel, rather than holes and stand-off mounts. Cons are that it uses up an extra couple of the Arduino pins which were previously spares, and that you don't have the proportional joystick mode available for those first trimming flights. But joystick mode still works, bang-bang on the four buttons, so I guess you could do trimming flights by pulsing the buttons like an old reed set!

I've written the code. I've managed to include both versions in the same code - it detects if you've built the 'button joystick' version when you press the 'right' or 'bottom' buttons during the initial calibration. I won't upload the code though till I've finished assembling and wiring up the new box I've printed so I can test the code on both versions for any bugs.

By the way, I'm calling two of the buttons 'top' and 'bottom' to avoid any confusion as to whether 'up' means up in the menus - and therefore at the top, or 'up elevator' and therefore at the bottom!

I've now tested out the 'button joystick' version of the code and I've uploaded the new 'universal' version that has the 27 model memories and detects whether you've built the analogue or digital joystick option during calibration. The forum allows me to go back and edit my old posts, so I've uploaded the new version in both the posts where there was previously one - at the bottom of page one, and on page four where I added the extra model memories.

There's no point in sending the new version to your Arduinos if you already have them working - the new code won't do anything different. If you do send the new version to your Arduinos then you'll probably have to 'recalibrate' and clear down the model memories - so jot down the numbers for any planes you've already set up before you do that.

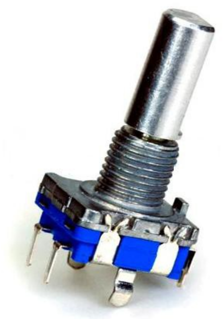
I've updated the sketch again. I've modified the original post at the bottom of page 1 to have the new version - but if your transmitter is already working you probably don't need to update.

The changes are:

- Fixed a small bug Ron found where it wasn't remembering your choice for 'Motor control by quick blip?' - you could switch it off for one flight but it would be back on the next time you cycled the power or changed to a different model.
- Made it so that certain screens such as model selection, model copying and model renaming are not visible when the motor is armed - makes it less likely to accidentally select a different model memory (which could result in a crash) when adjusting centre positions, etc. while flying.
- Small bug with motor soft start not soft-starting when the motor on microseconds were less than the motor off microseconds fixed.

Main change is that it now also supports a rotary encoder as an alternative to the analogue or button joystick for editing values. (A rotary encoder is a bit like the old 'dial' on some Futaba transmitters; some Spektrum transmitters have a 'roller' that does the same thing). You press the encoder to start editing a value, then wind the value up or down and press again to stop editing. When you're editing the model name the main 'tone' button moves on to the next character (or a long press moves back a character).

If you buy the cheap rotary encoders on ebay, they have three pins at one end and two pins at the other. Hold the encoder with the shaft pointing up and the three pins nearest to you. The left pin then connects to A0, the right pin to A1 and the centre pin to GND. The two pins on the other end are the encoder 'push button': one pin connects to GND and the other to D10.



You have to alter one line at the top of the sketch to compile the encoder version. You will see the second line down is:
```
// #define ENCODER
```
Remove the two slashes to make the line active and the encoder version will be compiled and uploaded. Leave the two slashes in place and you get the non-encoder version that works with the analogue joystick or the button joystick.
The encoder version will result in a neater case than either of the joystick versions - but of course the encoder is not much good for controlling a model in the air, so 'joystick mode' is absent if you decide to go for the encoder version. This may be preferable for some as it removes the possibility of 'cheating' (or others suspecting you may be cheating even when you're not)!
Everything else is pretty much the same as before.

I forgot to mention about the battery voltage calibration. Originally I was going to have a thing for calibrating that too, like the joystick, where you would enter a known voltage, measured with a meter and have the Arduino work out the right scaling constant. But then I figured that modern resistors are pretty accurate and by using a 22K and 10K for the battery divider we'd get a pretty accurate voltage without bothering with the calibration.
If you want a different voltage range or your resistors aren't exactly 22K and 10K then you can tweak the calibration by finding the line in the code:
```
eeprom::gd.batteryScale = 160;
```
...and editing it before sending it to your Arduino again. The number is the voltage, in tenths of volts, that you'd have to apply to the battery voltage resistor divider to get 5V at the centre. With 10K and 22K it works out to be 50 * (10 + 22) / 10 - so that's where the 160 comes from, and this gives a 16 volt input range which is plenty for any battery we might want to use - and it's the absolute maximum that the regulator chip on the Arduino can take without blowing up.
The code only writes the batteryScale value to EEPROM during the 'calibration' routine, so if you do change it you have to recalibrate. I know the encoder and the push button joystick don't really need calibrating, but it's handy to do it anyway as otherwise the model names and default values for all the unused models will have ridiculous non-standard values.

If anyone experiments with different sorts of encoder/scrollwheel, remember that swapping the two wires that connect to A0 and A1 over will reverse the direction of operation.
If you're not getting 'one count per notch' there is a constant in encoder.h By default this is set to 4, as the encoder that I used had one complete cycle of 'gray code' (and hence four edges to count) per notch. But I tried another encoder that only had half a cycle per notch, so with that one I had to set the constant to 2. There may be other encoders where the appropriate value would be 1 - if it's not working like you want, try changing the value - probably in powers of 2 so 1, 2, 4, 8, ... and see if that makes things better.
If you're not getting enough counts per revolution of the device you can alter the multiplier in the sketch. With most of the quantities that have a big range of adjustment, if you keep turning the encoder in the same direction without stopping for long, then after a few turns it shifts into high gear and counts in tens instead of ones. When you want to go back to counting in ones, you can either wait a couple of seconds, or just reverse the direction for a couple of clicks - both of these actions switch it back into low gear. Find the lines that say something like encoder::setMultiplier(10); and change them to 20 or 50. You could go to 100 I suppose but the sketch positions the cursor on the 'tens' column when you're in high gear, so it would look wrong going in steps of 100 unless you alter that too.
I've attached to this post an updated version with dual rates and some other new features.
Important: This version still has 27 model memories, but they're stored in a different format now, so if you upgrade an existing transmitter you'll lose all your model settings! If you have any models that have been carefully tuned to your preference, then write down all the numbers so you can re-enter them for this version.

Important: The joystick calibration and EEPROM reset pin is now A5, not D9. D9 is now used for telemetry (see next post). To calibrate and reset after upgrading switch on with A5 linked to GND. The link should be removed before switching on for a second time, or you'll lose any changes you've made.

I'll cover the telemetry options in the next post, but don't worry if you're not using a FrSky DHT module or you don't want to connect up the telemetry - all the telemetry 'pages' are invisible unless and until FrSky telemetry data is detected on pin D9.

Main changes:

- Dual Rates: for each of the three positions for rudder and elevator (centre and the two travel limits) you can now specify different positions for power on and power off. There is a 'G' setting and a 'P' setting for each. I think of them as 'Glide' and 'Power' settings, but you can think Giant/Pixie or Grand/Petite or whatever if you have a glider where 'Power' doesn't make sense.
- Servo Sweep mode: for servo testing (also useful for range checks). When activated the servos begin to sweep using the 'G' settings for the travel limits - these may be too small to see easily from a distance so you can use the joystick or encoder to wind up the amount of travel in several steps (indicated from 1 to 9). On the '9' setting the servos make the full travel (from 800 to 2200 microSeconds) - this may cause the surfaces to over-travel on some models, so be careful and advance the travel just one notch at a time.
- Four presses for 'kick down' elevator. If you don't want to risk this occurring by accident you can always set the elevator down positions to be the same as the centre ones.
- Inactivity alarm - if you leave the transmitter on for three minutes without moving anything, an audible alarm sounds.

I also fixed a couple of very minor bugs I found - but I've probably introduced some fresh ones too - so you may want to hold fire and let some other people test out the changes before you upgrade, unless you're keen!

If you have a FrSky DHT module and telemetry-enabled receivers, you can now monitor the received signal strength and the battery voltage of your model. There are audible alarms if the model is in danger of getting out of range or if the model battery voltage is getting low.
You connect the Txd output of your transmitter module to the D9 pin of the Arduino through a 100K resistor. I've tested without the resistor and didn't find any problems, but the Txd output does swing over a bigger voltage range (negative and positive) than a raw Arduino pin is supposed to take. Atmel who make the chip that the Arduino uses say it's quite okay to connect a voltage source with a big range to the pins as long as an in-line resistor is fitted to limit the current to a maximum of one milliamp. In one of their data sheets they show a circuit where mains voltage is wired to an input pin through a one meg resistor!
You can use any resistor from 10K up to about 100K - like I said I even tried it without any resistor at all and no damage occurred, but I don't really recommend that.
You need to bind to the model and have the model switched on to see the telemetry pages. If you have some models that don't have telemetry then the pages won't appear for those models.
You can monitor the RSSI level (received signal strength indication) This is 100 or more when the model is close. It drops away quickly as the distance increases, but then drops off much more slowly after 100 yards or so. The alarm goes off at 45, which is the default for the Taranis. You should still have control of your model till the RSSI drops to 40 or below. The difference between 45 and 40 doesn't sound much but when you get down to 45, you probably have to move the model at least twice as far away (in the air) to get down to 40.
You can monitor the A1 voltage - this is the receiver's internal voltage so it makes sense to use that on I.C. models or gliders as that will be the voltage of the model's on-board battery. For electric models the receiver just sees the BEC output voltage of the speed controller (usually about five volts) and although this is somewhat useful, what you really want to monitor is the main power-pack voltage.
This is where A2 comes in. The telemetry receivers (usually) have a pin labelled 'AD2' or similar for monitoring the power-pack. The range of AD2 is only 3.3V so you need an external divider circuit to drop the power pack voltage down to 3.3V maximum. FrSky sell

these divider circuits or you can make your own from a pair of resistors. I usually use a 47K to the battery positive and 10K to the battery negative - the two resistors are joined and the centre point connects to AD2. This works for 2, 3, or 4-cell LiPo packs - it gives a monitoring range of 3.3 x (47 + 10) / 10 = 18.81 volts for the battery. If you want to use five or six cell packs then you need to increase the 47K appropriately.
On the transmitter there is a 'Model battery scaling' page. Set the voltage to 0.0V if you wish to monitor the internal (A1) battery voltage for the alarm. The scaling is fixed for the internal A1 range, so the displayed A1 voltage should automatically be correct.
If you wish to monitor the external (AD2) voltage, set this scaling value to the input range (18.8 V in the above example). You can also just monitor the 'Telemetry A2' page and wind the scaling value up or down until the displayed 'A2' voltage is correct for your battery.
Once you've set the 'scaling' value to either zero or the appropriate range for A2, there is a separate page, 'Model battery alarm', where you set the voltage limit for the alarm to display and sound.
Hope that's clear enough. I'll try to answer any questions if anyone is confused.

For the oled display, it might be best to search through the main file and find all the lcd.setCursor( ) instances.
The format of all those is lcd.setCursor(column, row); so as it's only a two line display the second parameter is either 0 for the top row or 1 for the bottom row.
You could change all the instances of lcd,setCursor(??, 0); to lcd.setCursor(??, TOP_ROW); and similarly (??, BOTTOM_ROW) for the (??, 1) instances.
Then up near the top of the program you could have some conditional compilation macros like:

```
#define OLED
#ifdef OLED
#define TOP_ROW 1
#define BOTTOM_ROW 0
#else
#define TOP_ROW 0
#define BOTTOM_ROW 1
#endif
```

You might want to use TR and BR instead of TOP_ROW, BOTTOM_ROW to keep the lines shorter. Then you'd only need to comment out the #define OLED line to swap between the two versions.
If you swap the I/O pins around you might need to make some slightly tricky edits to the interrupt functions. The program uses pin change interrupts to decode the 'tone' push button, the rotary encoder pulses and the telemetry input from the FrSky. The Atmel chip provides separate interrupt vectors for pins 2 and 3 - and these can be enabled to trigger on +ve or -ve edge transitions or either. For the remaining pins the interrupts are shared across each port (8 pins) and you set a mask to enable which pins are enabled to trigger the interrupt (you can't select +ve or -ve for these - just any change).
The way I chose the pins, the rotary encoder, telemetry, and tone button are all on separate interrupt vectors - if you reassign the pins so that, say, the encoder and FrSky telemetry share the same port then you'll need some extra checks in the interrupt code to determine what event caused each interrupt.
If you're just flipping pins around within the same port or only changing pins that don't use interrupts then it will be easier.