

RC Joystick NG² Manual

(last update: 1 Oct 2006)

GENERAL DESCRIPTION

RC Joystick NG² is a little project to build an interface to connect a Radio Controller Transmitter to your computer via USB port. You can connect the radio to the interface in two way: **PPM/PCM mode**: use the PPM or PCM output (buddy box connector); **ADC mode**: uses a direct connection to radio potentiometers/switches doing an Analog to Digital Conversion. PPM support should allow to connect any radio equipped with PPM output. PCM support is limited to Sanwa/Airtronics PCM1/2 and Futaba PCM 1024. Note however that as far as we know at this time any PCM radio can also send PPM.

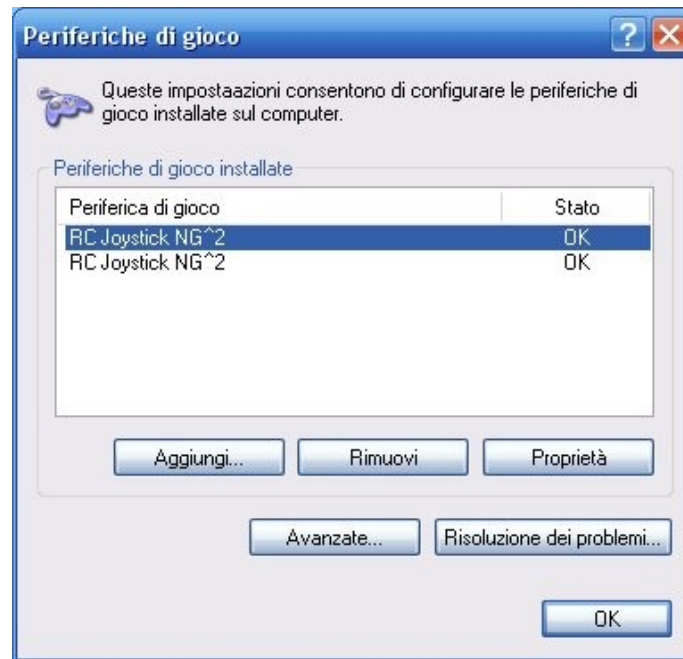


The SMD version of the interface

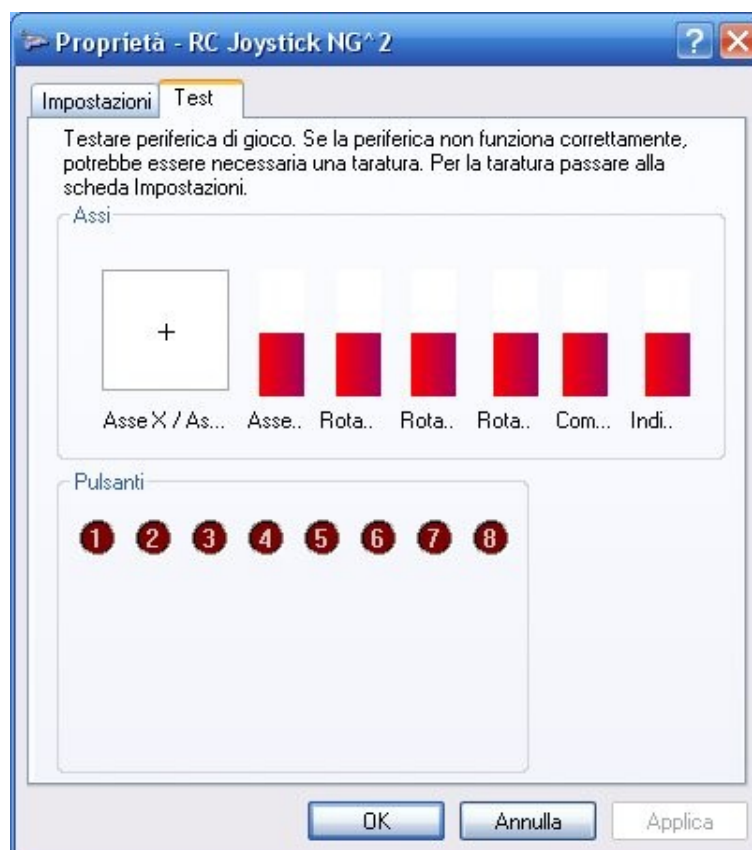
With this interface your favourite RC or flight simulator will see the RC TX as two traditional joysticks (instead of one as in RC Joystick / RC Joystick NG), now with 8 analog axes (up to about 5000 points resolution each) and 8 buttons each for a total of 16 analog axes and 16 buttons; 32 joystick controls can be assigned to a maximum of 12 radio channels: normal radios have less than 12 channels and PPM and PCM modulations doesn't allow more than 8/10 channels at the moment; however all these assignable joystick controls should gives better assignment versatility.... These assignments are stored inside PIC (non-volatile) eeprom and configurable by user directly via USB using a little utility written in C and *LibUSB-Win32* filter driver (Windows) or *libusb* (Linux, FreeBSD, NetBSD, OpenBSD, Darwin/MacOSX) or also by directly editing the pic eeprom data. This description refers to new 8 axis per joystick version; however for those who prefer the old 4 axis per joystick version it is still available in the software package (you can find there both compiled binaries and the source is compileable in both versions by just changing a define: see **THE SOFTWARE** section for further details).

The device enumerates as a USB Human Interface Device (HID) so no driver should be needed for normal use. Only if you want to change those assignments via USB you need to install the additional filter driver (see **THE SOFTWARE** section for further details). The new 8 axes per joystick version is a full speed device, the older 4 axes per joystick a low

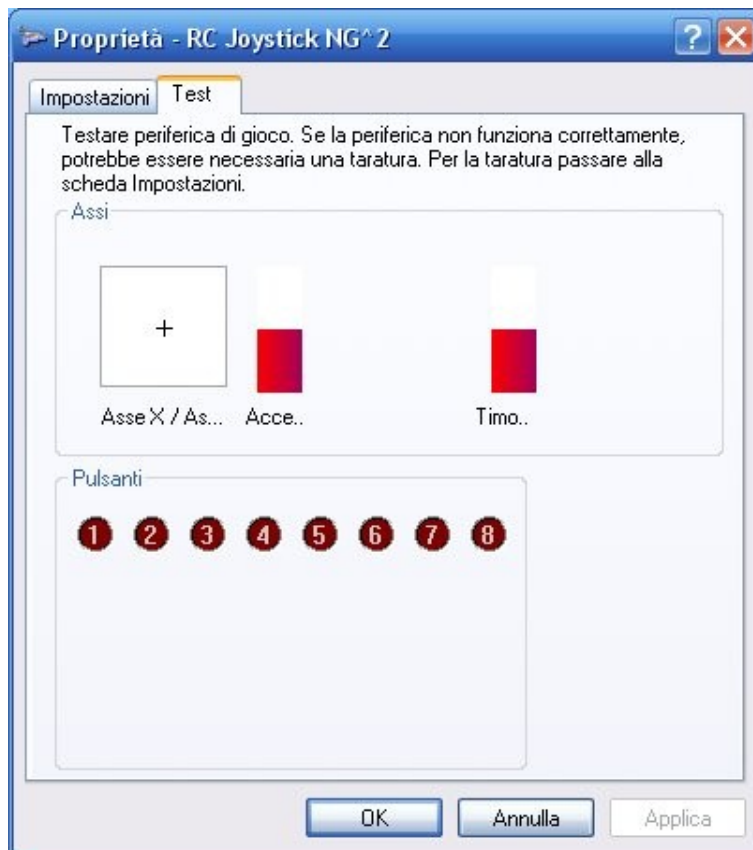
speed one.



*How the peripheral appears inside
Windows Control Panel -> Game Controller: 2 joysticks*



*8 axes (and 8 buttons) per joystick version:
properties of each one of the 2 joysticks*



*Older 4 axes (and 8 buttons) per joystick version:
properties of each one of the 2 joysticks*

Sorry... the figures above are from italian version of WinXP and it seems that Home Edition doesn't allow to change the language :-)

We are sure you understand that we are not responsible in any way if some smoke begins to come out from your computer and/or radio transmitter ;-) however read carefully the DISCLAIMER below.

DISCLAIMER

The package of Software and Hardware schemes you can download from our pages is provided "as is" without any guarantees or warranty. Although the authors have attempted to find and correct any bugs in the package, they are not responsible for any damage or losses of any kind caused by the use or misuse of the package. The authors are under no obligation to provide service, corrections, or upgrades to this package.

THIS SOFTWARE AND HARDWARE IS PROVIDED ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF WHAT YOU FIND ON OUR PAGES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

QUICK STEP BY STEP GUIDE

This is what you need to have the interface working on Windows:

- Build the hardware as described in **THE HARDWARE** section below. You can choose from a SMD and a Through Hole version. Double check it (especially USB connection and pinout)... after all you have to connect it to your radio controller and your PC!! ;-)
- Download the software package *rcjoyX.X.X.zip*; the package should unzip inside the top level directory *rcjoyX.X.X* (X.X.X is the version of the software).
- Program the PIC with the firmware (*rcjoyX.X.X\bin\pic\Xaxes\rcjoyX.X.X.hex*) as described in section **Programming the PIC** below; don't connect to USB while the device is connected to the programmer!! (read **Programming the PIC** NOTE)
- Disconnect the device from your programmer!! (read **Programming the PIC** NOTE) and connect it to a USB port. At this point the **Windows** should recognize (we hope ;-)) the hardware and automatically install the HID driver. Note that it could take a while to complete. If all goes ok under Control Panel -> Game Controllers you should see two new joysticks both simply named *RC Joystick NG^2*. The first in the list should be the one referred here as Joy1, the other Joy2. On most recent **Linux** distributions the device will be recognized as soon as you connect it to a USB port (if not you could need to manually load a kernel module... try *modprobe joydev*) and two new joysticks will be available as */dev/input/js0* and */dev/input/js1* (if you already have some other joystick connected the numbers could be different); to test them you can launch *jstest /dev/input/js0* and *jstest /dev/input/js1* (your distribution should provide it; on debian it's inside *joystick* package together with other related utilities).
- Connect the device to the radio in any way (probably you have to arrange some kind of connector and cable) referring to input signal connector pinout described in section **Connectors Pinout** below and looking for something like buddy box connector or PPM/PCM output description on your radio manual (**PPM/PCM mode**); if your radio have not a PPM/PCM output please refer to **Schematic** section to see how to directly connect its potentiometers/switches (**ADC mode**). At this point in **PPM/PCM mode** turning on the radio the led should also turn on; if it doesn't happen it probably means that the device was not correctly configured by the host (try also to disconnect, wait a while and reconnect) or the input PPM signal isn't read correctly (check the radio cable you have made, connectors, pinouts... and also refer to section **Known Issues**); in **ADC mode** the led will not turn on: it monitor only the PPM/PCM input signal (you can select **PPM/PCM mode** or **ADC mode** setting *Modulation* parameter with *set_assignments* utility; see **Assignments** below).
- NOTE: when you connect the radio to this interface you should remember to disable the transmission stage. This help to avoid the risk of interferences with other radios working near to you (or even with some electronic devices in the house as happened to us). Moreover the radio with transmission disabled request only a few percent of the power requested in normal use: the radio battery should last many many ours in this way. Refer to your radio manual on how to do that. With our Graupner MC-12 we simply remove the crystal.

The following steps are needed if you want to change default assignments table on the fly via USB (refer to section **THE SOFTWARE** below and in particular **Assignments**; note that when you program the firmware the eeprom will also be filled with default assignments, that are also the same you find below in this manual and in the file

set_assignments.txt provided inside the software package; if you don't need to change them you can ignore the following steps):

- Go to *rcjoyX.X.X\bin\windows* and install the filter driver *libusb-win32-filter-bin(...).exe* (for linux you need to have *libusb* installed)
- Go to *rcjoyX.X.X\bin\windows\Xaxes* directory using windows file explorer or the command line shell (for linux inside *rcjoyX.X.X\bin\linux\Xaxes*)
- Edit *set_assignments.txt* (or a different text file if you prefer) to suite your needs referring to section **Assignments** below.
- Launch *set_assignments*; you have the option to specify as a command line parameter the text input file you want to use if different from the default *set_assignments.txt*; *set_assignments* will also verify that eeprom is correctly written and reset the device to immediately use the new assignment settings; and because they are written in eeprom, it will remember them even if disconnected (until *set_assignments* is launched again or the entire firmware is reprogrammed). Note that if *set_assignments* exit with a parsing error you can be sure that nothing has been written to PIC eeprom yet: it begins to write eeprom only when parsing of input file is completed succesfully.
- If you want to check what assignments are saved into PIC eeprom you can use *get_assignments* for saving them to *get_assignments.txt* (or the text file you can specify as a command line parameter). Note that *get_assignments* doesn't ask before overwriting an existing destination file to avoid continuous annoying prompts... but if you don't like that refer to section **Computer Sources** below. The format of *get_assignments.txt* is the same readable by *set_assignments*. It is also well commented by the program: each table line shows the assignment code of the relative channel and as a comment also the exhaustive explanation of the assignment (channel and the meaning of the assignment code); so you can use it also to verify if you have correctly choosen assignment codes.

Enjoy it :-)

Alessio & Andrea

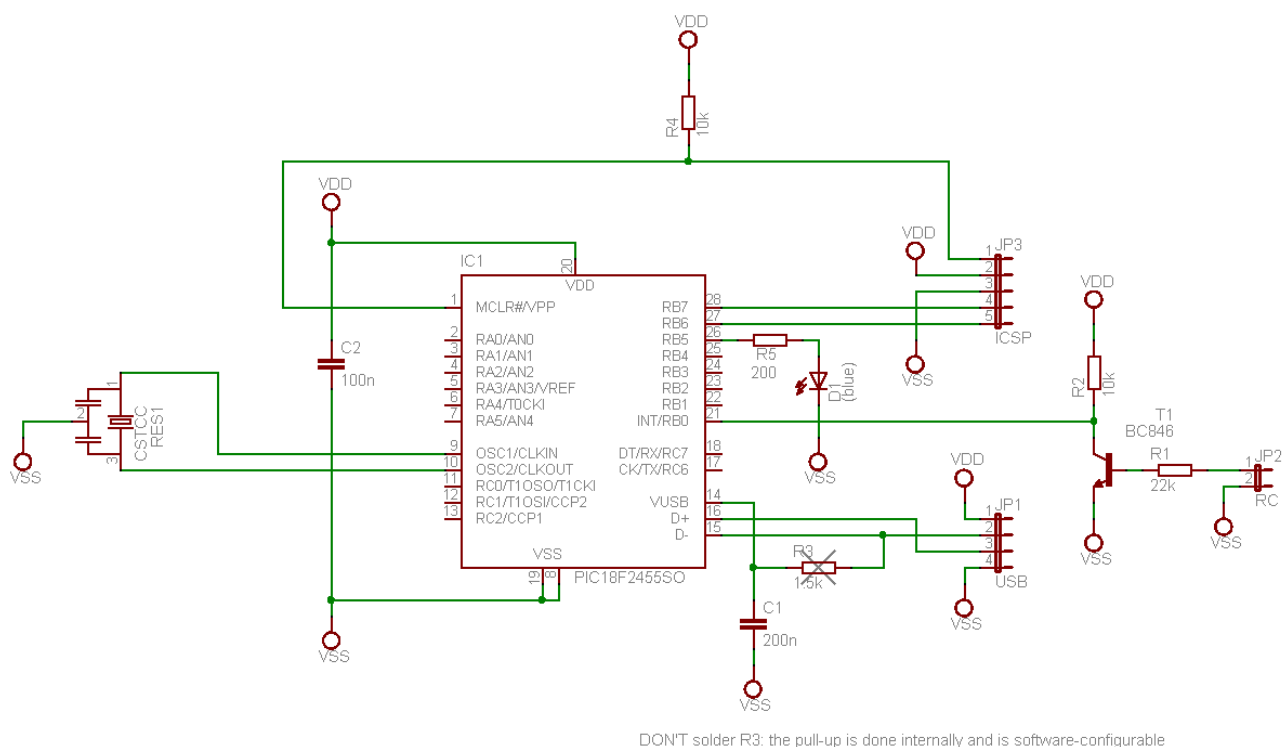
THE HARDWARE

RC Joystick NG² (like RC Joystick NG) can use the same PCB as RC Joystick: the only differences are in the resonator value, pull-up resistor R3 is no longer required and obviously in the completely different chip: PIC18F2455/2550 instead of PIC16F745/765; in fact from the hardware point of view these new chips are pin to pin compatible with the old chips, but require a different oscillator value, can be configured to work with internal pull-up instead of external resistor R3 (used before to be enumerated as a low speed device) and are completely different and not compatible from the firmware point of view; moreover they have flash program memory so can be programmed more than once and have also a non-volatile eeprom data memory used by this interface's firmware to store assignment settings.

Now the circuit comes in 2 versions: the original SMD 2 x 4 cm PCB and a new slightly grater (about 6 x 2.5 cm) through hole PCB for the ones who prefer this technology or find difficult to make SMD circuits at home. You can find the two versions inside the archives rcjoyng2_SMD.zip and rcjoyng2_TH.zip. The archive rcjoyng2_eagle.zip contains project folder for Cadsoft Eagle (we use Light Edition, the freeware non-profit limited version) you can find at <http://www.cadsoft.de/> .

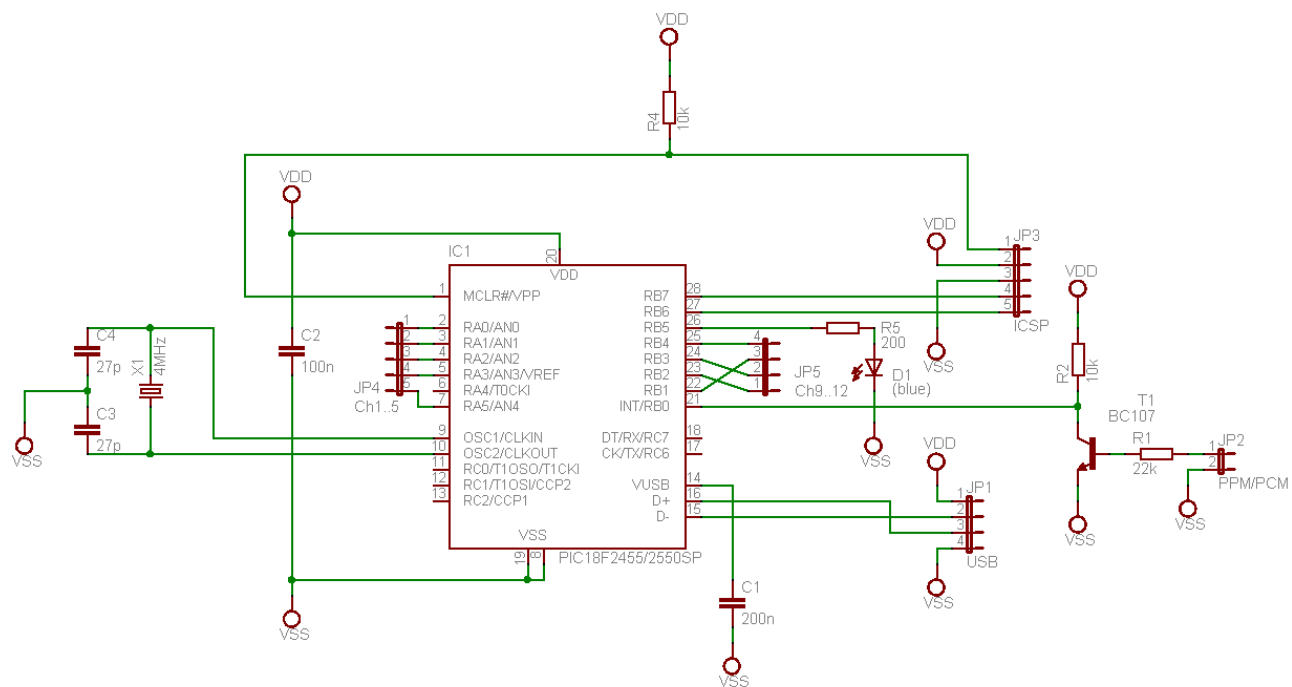
Schematic

Below you can see the schematic of the SMD version of the circuit; even if not specified here you can use both PIC18F2455/2550SO chips.



Schematic of the SMD version

The through hole version schematic is slightly different; only this one includes the analog inputs sockets (JP4 and JP5) for directly connecting radio potentiometers/switches (**ADC mode**):



R3 (pull-up from VUSB and D-) has been removed: the pull-up is done internally and is software-configurable

Schematic of the Through Hole version

The input PPM/PCM signal (used in **PPM/PCM mode**) from JP2 is connected to PIC INT/RB0 pin through a NPN transistor configured in common emitter mode (and then is inverted: this shouldn't cause any problem: see section **MODULATIONS** below). This input stage gives some problems with some JR radios (see also **Known Issues** section below for a solution).

For directly connecting old radios potentiometers/switches (**ADC mode**) we have provided specific connectors only on the Through Hole version of the PCB (see also **Connectors Pinout** below). In **ADC mode**, instead of reading from PPM/PCM input signal (RB0 pin), the software reads 12 channels Ch1..Ch12 from the PIC analog inputs pins AN0..AN11 through internal 10 bit (1024 levels) Analog to Digital Converter. Remember that on 28 pins device (18F2455/2550) AN5, AN6 and AN7 pins are not implemented so with this circuit Ch6, Ch7 and Ch8 will not be available: only Ch1..Ch5, Ch8..Ch12 corresponding to JP4 and JP5 pins; but the same firmware programmed into a 40/44 pin device (18F4455/4550) would read all 12 channels.

The voltage range converted by the ADC is from VSS = 0 to VDD = 5V; so you should verify that your radio feed the potentiometer with 5V: measuring the voltages of the two side connector of the pots referred to radio GND they should be 0 and 5V, while the cursor (pot center pin) should sweep between 0 and 5 while moving the stick; if that is ok all you have to do is: connect the ground of the interface (VSS on the ICSP) with the GND of your radio, each pot cursor (pot central pin) to a JP4 or JP5 pin and then turn on the radio; you could choose the arrangement of this connection depending on what radio brand and mode you want to emulate (refer also to channel usages list for some radio brands in section **Assignments** below); or you could simply choose your own private arrangement; but in both cases you should remember which stick pots is connected to each channel for

correctly configuring assignments later using the utility *set_assignments*. You can also try to connect radio switches to some channels... if for example you find that one of the switch pin voltage goes from GND to 5V... later you could assign it to an axis or button depending on simulator needs. With this kind of connection you have to keep your radio on when using with this interface (remember to remove the crystal or disable in any other way the transmission to avoid interferences and extend the life of the battery).

But in case your radio uses different voltage levels the things could be different: if the range is lower you could be still able to use the connection described above with a calibration inside control panel -> game controllers and some loss in resolution; but if the range is greater you should never connect the radio to analog inputs or the PIC could be damaged!! In this case, and also in case you don't want to keep your radio on, or you have an old useless radio, you could proceed with another type of connection: disconnect all the pots pin from the radio and connect them to the interface in the following way: the two side pins of each pot go one to VSS and one to VDD (you can take VSS and VDD from ICSP connector, refer to **Connectors Pinout**) and the pot center pin to one of the analog inputs as described before. Idem for a three pins switch. This type of connection has the advantage that the radio doesn't need to be turned on, then no need of recharging the battery... but also has the disadvantage that the pots pins should be disconnected from the radio before connecting to this interface to avoid possible damages.

Power supply (VDD/VSS) from USB (JP1) and from ICSP connector (JP3) are connected together inside the device so for security reasons you should never connect the device to both USB and Programmer (see section **Programming the PIC** below).

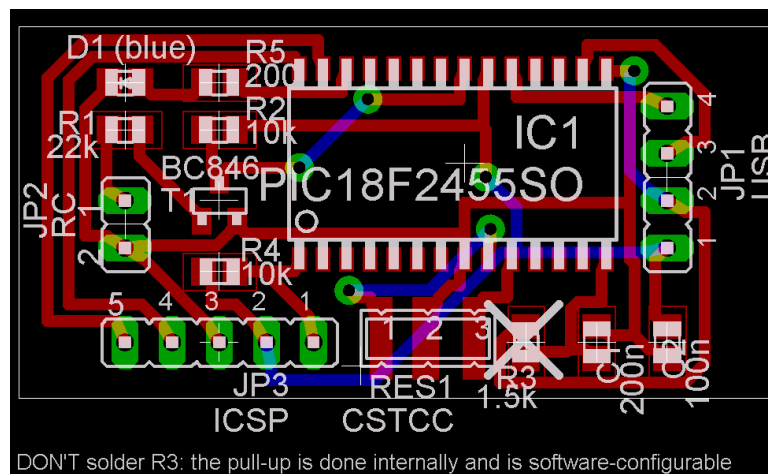
RB5 drives the led through a resistor; the value you see above ($R5 = 200\ \Omega$) is for blue led only. If you want to use lower voltage led (red, green, yellow...) you should use 1k resistor instead (or led could be destroyed... see also **Parts List** below). Led is on when the input signal is detected (and the host has successfully configured the device as a USB HID peripheral).

The pull-up of D- line is done internally and is software configurable so you don't need to solder R3. It is present here in SMD schematic and layout only because RC Joystick NG/NG² share the same PCB with old 16C745 equipped RC Joystick version that needs it (and also in case you prefer to modify the firmware sources disabling internal pull-up; see also **Parts List** below).

Resonator here is 4 Mhz (in old RC Joystick with 16C745 was 6MHz; see also **Parts List**).

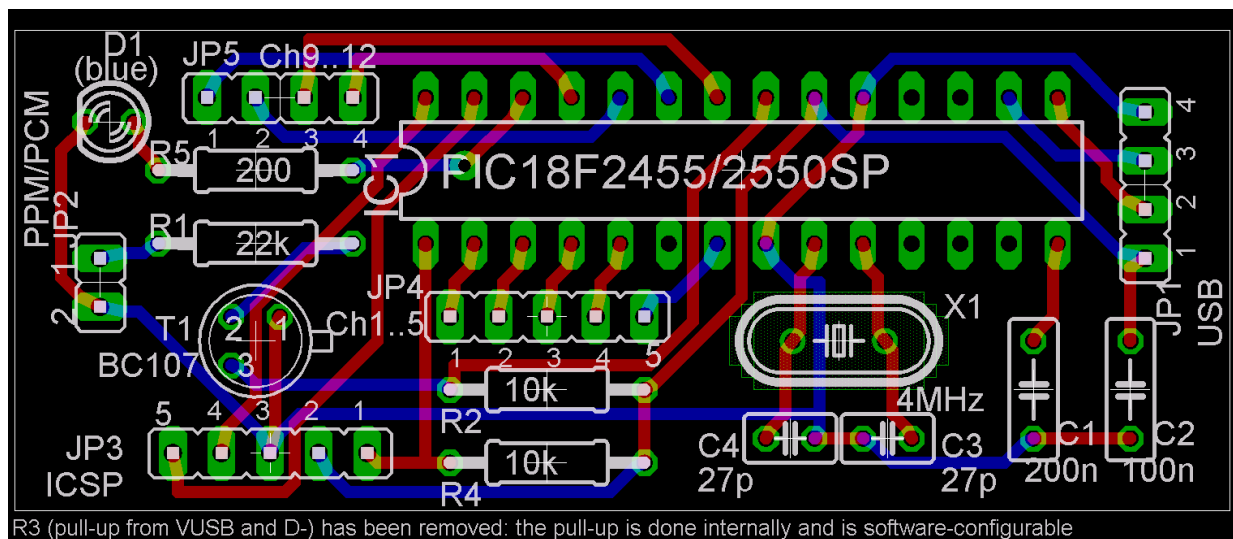
Layout

The layout you find below use SMD components to build a very small PCB (only 2 x 4 cm):



Layout of the SMD version

if you prefer you can use the through hole version (about 6 x 2.5 cm):

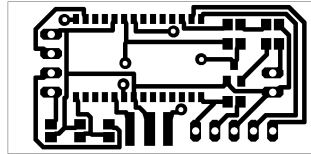


Layout of the Through Hole version

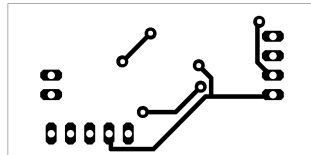
PCB

Below you find top and bottom sides of the PCB ready to print: scale 1:1, mirrored copper side so you can directly print them on transparencies and place ink side in contact with copper (photoresist covered) for exposing to UV (this prevent the UV light to diffuse under the ink through the transparency film causing less sharp edges).

The SMD version:

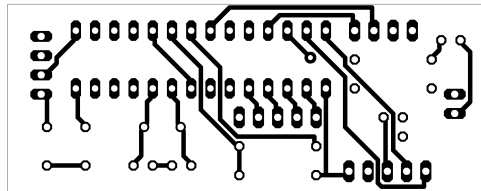


Top (SMD version; mirrored copper side; scale 1:1)

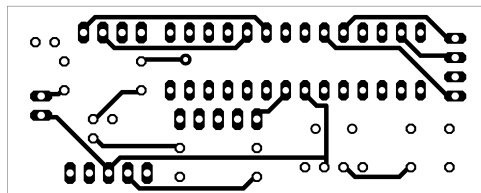


Bottom (SMD version; mirrored copper side; scale 1:1)

and the through hole one:



Top (Through Hole version; mirrored copper side; scale 1:1)



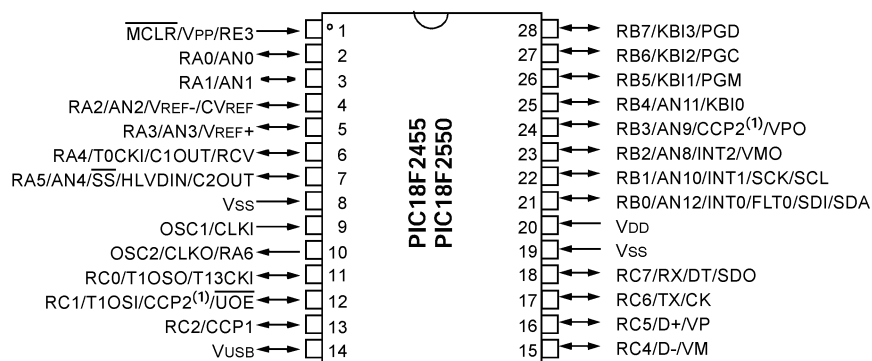
Bottom (Through Hole version; mirrored copper side; scale 1:1)

For the SMD version you have to solder 6 vias: you can do it by inserting a little piece of wire into the holes and soldering on both top and bottom sides.

For the Through Hole version you have to solder only one via but you should solder each pad of each component on both top and bottom sides (unless you are able to do metalized holes).

Connectors Pinout

- **JP1:** usb cable: 4 wires to the A type USB connector:
 1 = USB 1 (■ red wire = Vbus)
 2 = USB 2 (□ white wire = D-)
 3 = USB 3 (■ green wire = D+)
 4 = USB 4 (■ black wire = Gnd)
- **JP2:** RC transmitter cable: 2 wires from buddy box:
 1 = PPM Signal
 2 = GND
- **JP3:** ICSP connector: SIL 5 pin male connector:
 1 = VPP
 2 = VDD
 3 = VSS
 4 = RB7
 5 = RB6
- **JP4:** analog inputs connector for *ADC mode*: SIL 5 pin male connector
 1 = Ch1 (AN0)
 2 = Ch2 (AN1)
 3 = Ch3 (AN2)
 4 = Ch4 (AN3)
 5 = Ch5 (AN4)
- **JP5:** analog inputs connector for *ADC mode*: SIL 4 pin male connector
 1 = Ch9 (AN8)
 2 = Ch10 (AN9)
 3 = Ch11 (AN10)
 4 = Ch12 (AN11)
- PIC18F2455/2550: 28 pin PDIP or 28 pin SOIC



Parts List

- **SMD version:**
 - *RES1*: 4 MHz ceramic resonator (3 terminals with built in parallel capacitors for example muRata CSTCC4.00MG)
 - *C1*: 200 nF capacitor (0805)
 - *C2*: 100 nF capacitor (0805)
 - *R1*: 22 kΩ resistor (0805)
 - *R2, R4*: 10 kΩ resistors (0805)
 - *R5*: 200 Ω resistor, but only if D1 is blue !! (0805)
 - *IC1*: PIC18F2455-I/SO or PIC18F2550-I/SO (28-Lead SOIC)
 - *D1*: blue led; if you use other colors put R5=1k !! (0805)
 - *T1*: BC846 transistor or equivalent (SOT23)
- **Through Hole version:**
 - *X1*: 4 MHz crystal (for example HC49/S)
 - *C1*: 200 nF capacitor (polyester, grid 5mm)
 - *C2*: 100 nF capacitor (polyester, grid 5mm)
 - *C3, C4*: 27pF (ceramic, grid 2.5mm)
 - *R1*: 22 kΩ resistor (¼ W, 5%)
 - *R2, R4*: 10 kΩ resistors (¼ W, 5%)
 - *R5*: 200 Ω resistor, but only if D1 is blue !! (¼ W, 5%)
 - *IC1*: PIC18F2455-I/SP or PIC18F2550-I/SP (28-Lead PDIP)
 - *D1*: blue led; if you use other colors put R5=1k !! (3mm)
 - *T1*: BC107 transistor or equivalent (TO18)

Notes:

- The pull-up of *D-* line is done internally and is software configurable so you don't need anymore the 1.5k resistor *R3*: we have left the *R3* pads on the SMD PCB that is the same PCB as in RC Joystick / NG but you must not solder it unless you modify the firmware sources disabling internal pull-up; on the Through Hole version it has been removed.
- If you prefer to use low voltage leds (red, green, yellow...) instead of blue ones you have to replace *R5* 200 Ohm resistor with 1K one, or your led could be destroyed.

Assembling

Once you have soldered all components you have to find a way to connect it to USB and your radio: we have directly soldered to JP1 the final part of a USB cable and to JP2 a spiral audio cable with a mono plug (refer also to **Connectors Pinout** above). Then we have mounted a mono jack on the radio for connecting to buddy box socket inside. You may have to choose a different plug depending on your radio buddy box connector. You can see the result of the SMD version (we have not made the Through Hole one) in the photos below. We have chosen to solder the cables directly to the PCB and fix them with hot melt glue but if you prefer you can mount two SIL 4 pin and 2 pin connectors on the PCB and female connectors on the cables. Moreover we prefer to see the PCB "naked" :-)) but perhaps you would like to mount all in a little box... recommended to avoid external contacts and short circuits....



Really there is also an alternative way to mount it: the PCB is so small (4 x 2 cm for the SMD version and about 6 x 2.5 for the Through Hole one) that you could mount it directly inside your radio if you find a way to arrange a USB connector for the external cable and a pass-through connection between buddy box internal socket and external connector; the input impedance of this circuit should be high enough (>20k) to be almost transparent. If

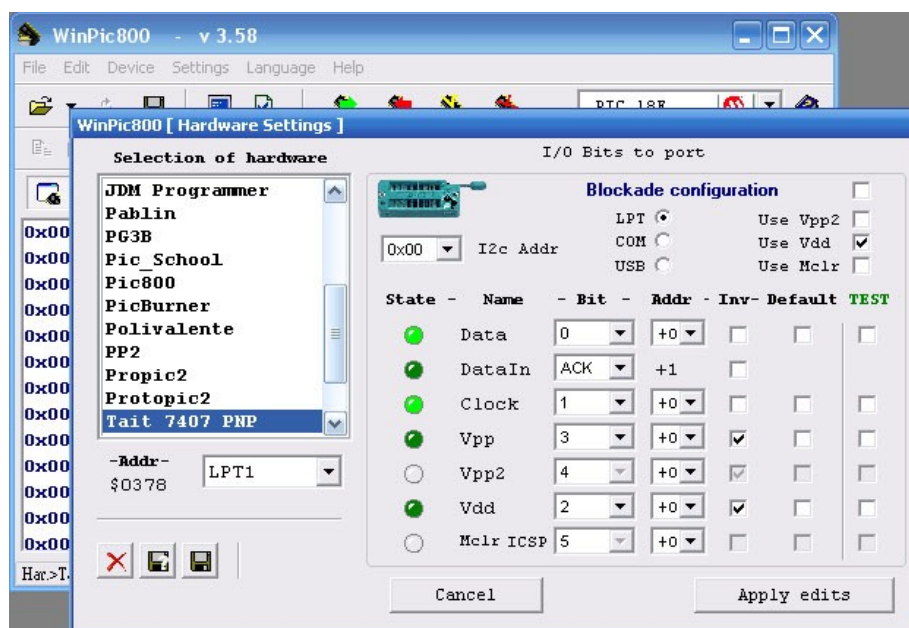
your radio has not PPM/PCM you could do it with ADC mode: if potentiometers/switches inside are fed with a 5V voltage you don't need to disconnect them from the radio; otherwise or in case you have an hold useless or malfunctioning radio you can transform it into a stand alone controller for PC. Obviously the disadvantage of internal mounting is that if you want to use more radios you need more interfaces...

Programming the PIC

Before using the interface you need to program the PIC with the firmware binary file *rcjoyX.X.X\bin\pic\Xaxes\rcjoyX.X.X.hex*. Xaxes stands for 8axes or 4axes that refers to the two versions of the firmware: 8 axes per joystick full speed device or 4 axes per joystick low speed device; simply choose the one you prefer...

You should be able to program the 18F2455/2550 with our PicProgrammer (you can find at <http://projects.qstep.net/> -> PicProgrammer) and WinPic800 software (you can find at <http://www.winpic800.com/>).

For setting WinPic800 to work with our PicProgrammer (or a "Tait classic" programmer) you have to go to "settings->hardware" select "Polivalente" in the "Selection of hardware" and "save as" Tait 7407 PNP; then UNCHECK "Blockade configuration" and modify the settings as showed in the picture below; when finished remember to check again "Blockade configuration" and click on "Apply edits":



For correct programming of eeprom data memory with WinPic800 you need also to uncheck the option: Settings -> Software -> .hex -> File .HEX -> Data 18Fxxx Address * 2:



Obviously if you prefer you can use your favorite hardware/software (in this case you should check before if it's compatible with 18F2455/2550)

The interface has an ICSP connector you can use to program the chip directly on circuit.

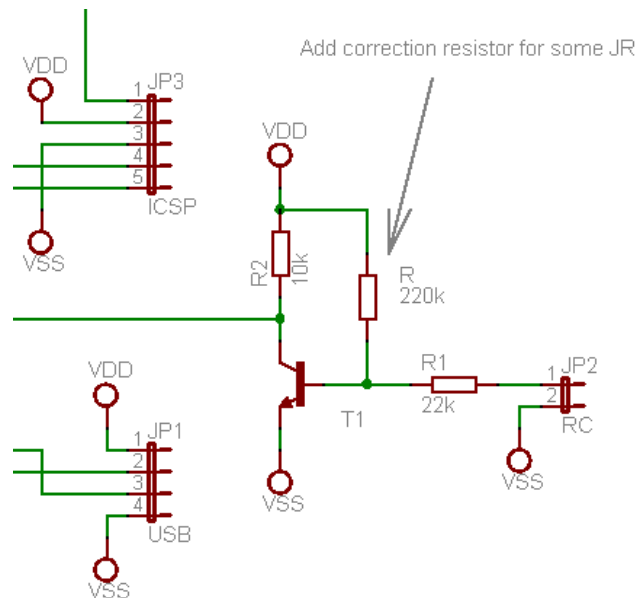
NOTE: Remember to DISCONNECT from USB port before connecting the ICSP to programmer and viceversa: USB and ICSP should NEVER be connected both at the same time to avoid possible damages to the programmer or computer!! (the 5V PC USB supply and 5V programmer ICSP supply will be connected together and probably they haven't exactly the same level...).

For the ICSP connector pinout see above **Connectors pinout**.

For more informations about ICSP pic programming and supported microcontrollers see Microchip ICSP Guide at <http://ww1.microchip.com/downloads/en/DeviceDoc/30277d.pdf>

Known Issues

Some JR owners have reported unstable behaviour of the input stage with their radios (irregular blinking of led... irregular joystick controls...). The following correction seems to work for this radios: you have to add a 220k resistor between VDD and the transistor base. We don't know really what causes the problem because we have never had one of this radio to test... but we can suppose that these radios could probably use some kind of open collector output... anyway this correction worked ok for all who reported the problem:



THE SOFTWARE

For this interface you need two softwares: the firmware for programming the PIC on the board and a program for your computer needed for configuring channels assignments directly via USB. In the zip archive *rcjoyX.X.X.zip* you'll find sources and binaries of both. *rcjoyX.X.X.zip* should unzip to a top level directory called *rcjoyX.X.X* (X.X.X is the software version).

PIC Firmware Binary

The firmware binary (ready for programming the pic) is:

rcjoyX.X.X\bin\pic\Xaxes\rcjoyX.X.X.hex

where Xaxes stands for 8axes or 4axes that refers to the two versions of the firmware: 8 axes per joystick full speed device or 4 axes per joystick low speed device; simply choose the one you prefer...

PIC Firmware Source

Firmware source is inside *rcjoyX.X.X\source\pic*. For edit/compiling it using Microchip MPLAB IDE you need to do the following steps:

- open MPLAB IDE and create a new project specifying *rcjoyX.X.X\source\pic* as project directory and *rcjoyX.X.X* as project name.
- Referring to the project tree window add *rcjoy.asm* inside *Sources Files*; *ENGR2210.inc*, *rcjoy_defs.inc* and *usb_defs.inc* inside *Headers Files*; *18f2455.lkr* inside *Linker Scripts*.
- choose PIC18F2455 inside *Configure -> Select Device*
- select which version you want to compile by changing the *INTERFACE_TYPE* define inside *rcjoy_defs.inc*: 0 for 8 axes per joystick full speed device, 1 for 4 axes per joystick low speed device
- to compile.... *Project -> Build All*

Firmware for PIC18F2455/2550 is written in assembler and is based on Bradley A. Minch assembler framework you find at <http://pe.ece.olin.edu/ece/projects.html> (in particular lab2) with the addition of the PPM and PCM signal handling and returning data routines and some modifications to USB part (descriptors, vendor specific requests, and some adjustments to make it act as a compound and full speed device).

Thanks to Shaul Eizikovich, author of SmartPropoPlus, for PCM signal infos and help for testing; his project page: http://www.geocities.com/shaul_ei/SmartPropoPlus.html

Thanks to Rinaldo Dos Santos for Futaba PCM1024 recordings and help for testing.

This firmware uses PIC Timer1 (16bit) for measuring pulses duration (see also **MODULATIONS** below); this (together with proper Timer1 setting) allows to get up to about 5000 points of resolution per channel if input signal is PPM. Depending on its modulation the input signal is decoded to calculate each channel value and the corresponding value of the joystick control assigned to that channel; this assignment is

software configurable and stored into pic eeprom: see section **Assignments** below for details. If the assigned joystick control is a button a pressed button condition is returned if the corresponding channel value is greater than a threshold (see *BUTTON_THRESHOLD* define inside *rcjoy_defs.inc*).

Computer Binaries

Computer binaries for Windows are inside *rcjoyX.X.X\bin\windows\Xaxes* (for linux inside *rcjoyX.X.X\bin\linux\Xaxes*), where again *Xaxes* stands for 8axes or 4axes, choose the one that corresponds to the firmware version you want to program: 8 axes per joystick full speed device or 4 axes per joystick low speed device... If you use **Windows** you first need to install the filter driver *libusb-win32-filter-bin(...).exe* inside *rcjoyX.X.X\bin\windows* (it comes from *LibUSB-Win32* project you can find at <http://libusb-win32.sourceforge.net/>); for **Linux** you need to have *libusb* installed (<http://libusb.sourceforge.net/>). After that you can edit *set_assignments.txt* to suite your needs and then connect the interface to an USB port and use *set_assignments* to send them to the interface; the program also verify that the assignments are correctly written inside PIC eeprom and reset the device to use the new settings. If for some reason you need to read the assignments from interface you can use *get_assignments* that will write them to the file *get_assignments.txt*. If you prefer to use a different file name instead of the defaults you can specify it as a command line parameter (for ex.: *set_assignment nomefile*). See also section **Assignments** below for further details.

Commands below will display usage:

```
set_assignments -h (or --help, /h, /?)  
get_assignments -h (or --help, /h, /?)
```

Computer Sources

Computer sources are inside *rcjoyX.X.X\source\computer*: they work for both linux and windows: for compiling on **Windows** you need to install MinGW (<http://www.mingw.org/>) and the *LibUSB-Win32* filter driver *libusb-win32-filter-bin(...).exe* in *rcjoyX.X.X\bin\windows* and then, supposing that *mingw\bin* directory (where *gcc.exe* is located) is in your path environment variable, launch ***gcc_compile.bat*** that contains the following commands:

```
gcc set_assignments.c c:\Programmi\LibUSB-Win32\lib\gcc\libusb.a -o set_assignments.exe  
gcc get_assignments.c c:\Programmi\LibUSB-Win32\lib\gcc\libusb.a -o get_assignments.exe
```

set_assignments.c and *get_assignments.c* also contains the following include:

```
#include c:\Programmi\LibUSB-Win32\include\usb.h
```

if something goes wrong also check that *libusb.a* and *usb.h* are in these locations. To compile on **Linux** you need to have installed *libusb* (together with devel package: usually -dev or -devel, for example on debian linux we have to install *libusb* and *libusb-dev* packages) and then simply run ***gcc_compile.sh*** that contains the following commands:

```
gcc set_assignments.c /usr/lib/libusb.a -o set_assignments  
gcc get_assignments.c /usr/lib/libusb.a -o get_assignments
```

set_assignments.c and *get_assignments.c* also contains followin include:

```
#include <usb.h>
```

You can find *LibUSB-Win32* at <http://libusb-win32.sourceforge.net/> and *libusb* at <http://libusb.sourceforge.net/>

Before compiling the two utilities verify that the `INTERFACE_TYPE` define inside the two source files reflects the firmware version you are using (8 axes per joystick or 4 axes per joystick).

set_assignments.c parses the file *set_assignments.txt* (or the one specified on command line) and uses *libusb* function `usb_control_msg` to send the interface vendor requests for writing eeprom bytes, reading them for verifying and then resetting the running firmware to use the new values. See also **Assignments** below.

get_assignments.c use *libusb* function `usb_control_msg` to send the interface vendor requests for reading eeprom bytes and then writes them to the file *get_assignments.txt* (or the one specified on command line; note that this program doesn't ask before overwriting an existing destination file; if you want it to ask before overwriting you have to uncomment the `PROMPT_OVERWRITING` define. See also **Assignments** below.

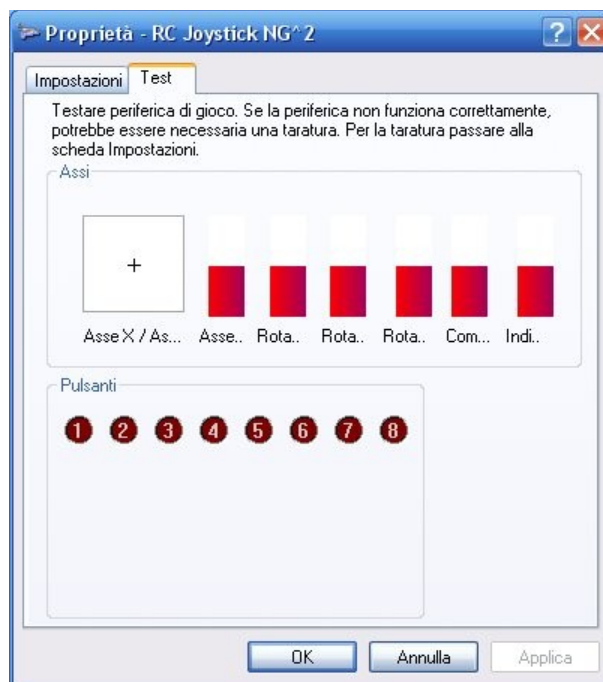
Assignments

The 8 axes per joystick version of the interface can read a maximum of 12 radio channels and a total of 32 joystick controls (24 for 4 axes per joystick version) can be assigned to each one of these channels: normal radios will have less than 12 channels and PPM and PCM modulations rarely allows more than 8 channels at this time; anyway all these assignable joystick controls serves to give a good assignment versatility....; the assignments are software configurable and stored inside the PIC 256 byte eeprom: from 1 to 18 different assignments tables can be prepared into eeprom to suite different radios/simulators needs. At any time it is possible to change which table to use by simply changing last eeprom byte (TableSelect byte). The first byte of a table is the Modulation code byte, that specify the type of modulation **PPM/PCM** used by the radio or 0 in case youre radio is not equipped with PPM/PCM and you are going to use the **ADC mode**: directly connect the potentiometers/switches of your radio to analog input AN0..AN11 (JP4, JP5) instead of using PPM/PCM input; the following 12 bytes are the assignment codes of each channel. Assignments codes are ordered inside each table from Ch1 to Ch12.

Modulation codes have the following meanings (see also **MODULATIONS** below):

- 0 (0x00) = ADC mode: Ch1..Ch12 from PIC pin AN0..AN11 (JP4, JP5), ADC range VSS..VDD (0..5V)
- 1 (0x01) = PPM/PCM mode: PPM
- 2 (0x02) = PPM/PCM mode: Sanwa/Airtronics PCM1
- 3 (0x03) = PPM/PCM mode: Sanwa/Airtronics PCM2
- 4 (0x04) = PPM/PCM mode: Futaba PCM 1024

Assignment codes of joystick controls depend on what version you decide to use: for the 8 axes per joystick version we have for each one of the two joystick (note that the figures below are screenshot of an italian version of WinXP and unfortunately WinXP Home Edition does not seem to allow to change the language so we can't show them in english)



*8 axes (and 8 buttons) per joystick version:
properties of each one of the 2 joysticks*

the relative assignment codes are:

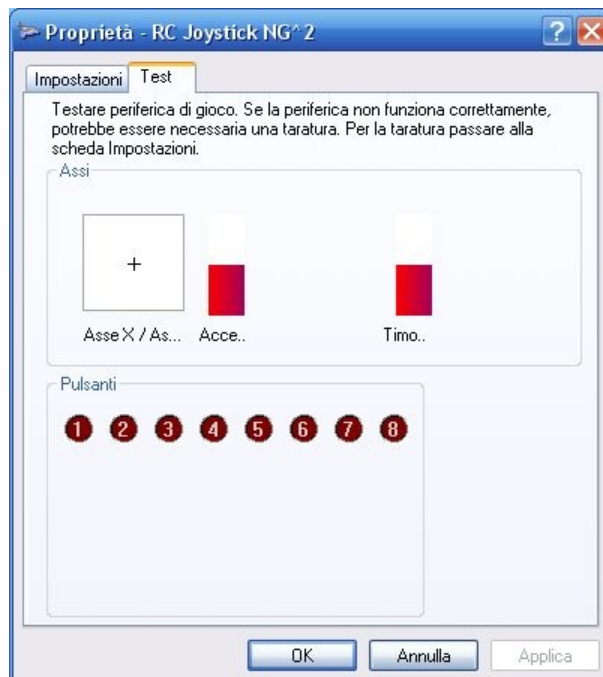
```
0 (0x00) = none (relative channel ignored)

1 (0x01) = Joy1 X axis
2 (0x02) = Joy1 Y axis
3 (0x03) = Joy1 Z axis
4 (0x04) = Joy1 RotX axis
5 (0x05) = Joy1 RotY axis
6 (0x06) = Joy1 RotZ axis
7 (0x07) = Joy1 Dial axis
8 (0x08) = Joy1 Slider axis
9 (0x09) = Joy1 Button 1
10 (0x0a) = Joy1 Button 2
11 (0x0b) = Joy1 Button 3
12 (0x0c) = Joy1 Button 4
13 (0x0d) = Joy1 Button 5
14 (0x0e) = Joy1 Button 6
15 (0x0f) = Joy1 Button 7
16 (0x10) = Joy1 Button 8

17 (0x11) = Joy2 X axis
18 (0x12) = Joy2 Y axis
19 (0x13) = Joy2 Z axis
20 (0x14) = Joy2 RotX axis
21 (0x15) = Joy2 RotY axis
22 (0x16) = Joy2 RotZ axis
23 (0x17) = Joy2 Dial axis
24 (0x18) = Joy2 Slider axis
25 (0x19) = Joy2 Button 1
26 (0x1a) = Joy2 Button 2
27 (0x1b) = Joy2 Button 3
28 (0x1c) = Joy2 Button 4
29 (0x1d) = Joy2 Button 5
30 (0x1e) = Joy2 Button 6
31 (0x1f) = Joy2 Button 7
32 (0x20) = Joy2 Button 8

NOTE: negating a code reverses the relative channel
```

For the 4 axes per joystick version we have instead for each one of the two joystick:



*Older 4 axes (and 8 buttons) per joystick version:
properties of each one of the 2 joysticks*

and the relative assignment codes are:

```
0 (0x00) = none (relative channel ignored)

1 (0x01) = Joy1 X axis
2 (0x02) = Joy1 Y axis
3 (0x03) = Joy1 Throttle axis
4 (0x04) = Joy1 Rudder axis
5 (0x05) = Joy1 Button 1
6 (0x06) = Joy1 Button 2
7 (0x07) = Joy1 Button 3
8 (0x08) = Joy1 Button 4
9 (0x09) = Joy1 Button 5
10 (0x0A) = Joy1 Button 6
11 (0x0B) = Joy1 Button 7
12 (0x0C) = Joy1 Button 8

13 (0x0D) = Joy2 X axis
14 (0x0E) = Joy2 Y axis
15 (0x0F) = Joy2 Throttle axis
16 (0x10) = Joy2 Rudder axis
17 (0x11) = Joy2 Button 1
18 (0x12) = Joy2 Button 2
19 (0x13) = Joy2 Button 3
20 (0x14) = Joy2 Button 4
21 (0x15) = Joy2 Button 5
22 (0x16) = Joy2 Button 6
23 (0x17) = Joy2 Button 7
24 (0x18) = Joy2 Button 8

NOTE: negating a code reverses the relative channel
```

Hex codes inside parenthesis will help in case of direct editing of the eeprom data but inside the file *set_assignments.txt* (see below) you have to use DECIMAL codes!!

NOTE that from version 3.1.3 you can also assign the reversed channel to a joystick control by negating the relative code. For negating a code inside *set_assignments.txt* simply put a minus sign in front of the decimal code (WITHOUT spaces in between, see examples below); if you prefer to directly change the codes in the data eeprom or in the default tables in the source remember to use one byte 2's complement hex values as negated code.

Table 1 (AssignTab1) fills first 13 bytes of eeprom, table 2 fills the following 13 and so on for a maximum of 18 tables. A TableSelect value of 1 means table 1 selected and so on.

You could change AssignTab1..18 and TableSelect directly using your programmer software or changing eeprom data byte declarations in the source, recompiling it and reprogramming the PIC. However the fastest way to do that is directly through USB with the program ***set_assignments.exe***. It requires that *LibUSB-Win32* filter driver is installed on your system (see section *Computer Binaries*). All you need to do is to edit *set_assignments.txt* and set your preferred assignments: you can define from 1 to 18 tables: for defining a table you have to write the relative label (for ex. AssignTab1) in a line and then up to 13 table elements (decimal) in the following lines, up to one value per line. First value will be the modulation code. Each of the following values will be relative to next channel, void lines are simply ignored and comments must begin with ";". If less than 13 elements are specified the others will be set to 0 by default inside the eeprom (that simply cause the relative channels to be ignored, or, in case of the modulation code, to read the channels not from the RB0/INT0 input signal but from pots connected to AN0..AN11 inputs). Table select byte is defined specifying the label TableSelect in a line and the value (decimal) in one of the following lines. AssignTab1..18 and TableSelect can be defined in any order inside the file. Every missing table will be filled with 0 inside the eeprom. If

missing, TableSelect will be 1 in the eeprom (table 1 selected). Obviously for the interface to work ok TableSelect should point to a defined table. However any table or table select error can't cause the interface to hang... only wrong assignments. Note also that *set_assignments* doesn't write anything to PIC eeprom until the parsing process of input file is completed successfully.

The possibility to write multiple tables was introduced for the ones who prefer (or are forced to because for example the utility *set_assignments* isn't ported yet on their OS) to directly change PIC eeprom with a programmer (in this way, once they have prepared the tables, they have to change only TableSelect byte). Also the ones who prefer to take all their settings in a single file can appreciate that. However, if you prefer, you can also take a single assignment table per file and use multiple files passing their names to *set_assignments.exe* as a command line parameter.

To make the assignments you could have to consider the usages of the 4 main channels of your radio; below you find these usages for some radio brands:

```
Futaba/Hitec:
  Ch1: Aileron
  Ch2: Elevator
  Ch3: Throttle
  Ch4: Rudder

JR/Graupner:
  Ch1: Throttle
  Ch2: Aileron
  Ch3: Elevator
  Ch4: Rudder

Sanwa/Airtronics:
  Ch1: Elevator
  Ch2: Aileron
  Ch3: Throttle
  Ch4: Rudder

Multiplex/Robbe:
  Ch1: Ailerons
  Ch2: Elevator
  Ch3: Rudder
  Ch4: Throttle
```

As an example here is the the assign tables from the *set_assignments.txt* provided in the software package for the 8 axes per joystick version:

```
; example 1:
; generic; first 8 channels assigned to Joy1 8 proportional axes,
; the remaining 4 to Joy1 first 4 buttons

AssignTab1
  1      ; Modulation
  1      ; Ch1
  2      ; Ch2
  3      ; Ch3
  4      ; Ch4
  5      ; Ch5
  6      ; Ch6
  7      ; Ch7
  8      ; Ch8
  9      ; Ch9
  10     ; Ch10
  11     ; Ch11
  12     ; Ch12

; example 2
```



```
; for FMS: with these assignments under Mapping / Calibration
; of Joystick interface (FMS menu: Controls -> Analog control...)
; we will see all the first eight channel of our radio correctly
; numbered and working. All you have to do is to assign whatever
; functions to them according to channel usages of your radio
; (above there is also a list of the channel usages of some radio
; brands)
```

AssignTab2

```
1      ; Modulation
1      ; Ch1
2      ; Ch2
3      ; Ch3
6      ; Ch4
8      ; Ch5
4      ; Ch6
17     ; Ch7
18     ; Ch8
0      ; Ch9
0      ; Ch10
0      ; Ch11
0      ; Ch12
```

; example 3

```
; futaba radio with non Interlink Realflight G2: first 4 channels
; mapped to 4 proper Joy1 controls and the following 4 to the first
; 4 buttons; with this assignments all we need is to go inside
; realflight controller calibration and choose RC Joystick NG^2
; with mode 2 selected; in this way we can also set the preferred
; stick mode (mode 1, 2, 3 or 4) directly inside the radio. If
; your Futaba has PCM1024 you can use PCM instead of PPM putting
; 4 in the Modulation field (Futaba PCM1024 modulation).
; For other radio brands we simply need to rearrange the order of
; the first 4 assignments (1 2 7 6) comparing the channel usages
; of our radio with futaba's (refer to the channel usages list
; above), moving each of the 4 assignment to the channel that has
; the same usage. For example for JR/Graupner we'll have 7 1 2 6.
```

AssignTab3

```
1      ; Modulation
1      ; Ch1
2      ; Ch2
7      ; Ch3
6      ; Ch4
9      ; Ch5
10     ; Ch6
11     ; Ch7
12     ; Ch8
0      ; Ch9
0      ; Ch10
0      ; Ch11
0      ; Ch12
```

; example 4

```
; ADC mode with Realflight G2: this example show the assignments
; in case we have connected 4 pots and 4 switches of an old radio
; not equipped with PPM or PCM output directly to analog inputs JP4
; and JP5 instead of using PPM/PCM input: in case we choose the
; connection arrangement with JP4 for emulating a Futaba mode 2
; radio (Ch1 = aileron = right stick left-right, Ch2 = elevator =
; right stick up-down, Ch3 = throttle = left stick up-down, Ch4 =
; rudder = left stick left-right) and connected switches 1..4 to
; Ch9..Ch12 (JP5), the only thing we have to do as in the previous
; example is to go inside realflight controller calibration and
; choose RC Joystick NG^2 with mode 2 selected.
```

AssignTab4

```
0      ; Modulation
1      ; Ch1
2      ; Ch2
7      ; Ch3
6      ; Ch4
0      ; Ch5
0      ; Ch6
0      ; Ch7
0      ; Ch8
```

```

9      ; Ch9
10     ; Ch10
11     ; Ch11
12     ; Ch12

```

```

; example 5
; FlightGear flight simulator; the following assignments let
; flightgear recognize up to 12 proportional channels using
; both Joy1 and Joy2 axes; run fgjs utility inside FlightGear
; bin directory for setting joystick control usages; this
; will make the two files js0.xml and js1.xml that you could
; rename Joy1 and Joy2 and put inside a directory rcjoy inside
; data\input\joysticks; then you should declare them inside
; Joystick.xml using numbered way instead of named one: both
; Joy1 and Joy2 under windows are named RC Joystick NG^2 (and
; for now we have not found a way to give different names to
; them) so they are not distinguishable using named declaration;
; example of these declarations:
; <js n="0" include="Input/Joysticks/rcjoy/Joy1.xml"/>
; <js n="1" include="Input/Joysticks/rcjoy/Joy2.xml"/>
; if you have other joysticks and don't know what number to put
; use js_demo utility that comes with FlightGear inside bin dir
; some channel could need to be reversed (here or inside the xml)

```

```

AssignTab5
1      ; Modulation
1      ; Ch1
2      ; Ch2
3      ; Ch3
4      ; Ch4
6      ; Ch5
8      ; Ch6
17     ; Ch7
18     ; Ch8
19     ; Ch9
20     ; Ch10
22     ; Ch11
24     ; Ch12

```

```

TableSelect
1

```

Here is another file example, perhaps it could seem less rigorous but it is as correctly formatted as before; it shows that formatting requirements are not so rigorous and also shows how to reverse a channel or tell the firmware to ignore some channels:

```

; this is another
; correct example

```

```

AssignTab7    ; reads only first 10 channels, the remaining 2 channels will be ignored

1            ; Modulation: PPM

; Ch1 to Ch8 assigned to Joy1 analog channels (JR/Graupner optimized arrangement)
1            ; Ch1
2            ; Ch2
4            ; Ch3
-3           ; Ch4: NOTE that here Ch4 is assigned to Joy1 Z Axis (code 3) but REVERSED (-)
5            ; Ch5
6            ; Ch6
7            ; Ch7
8            ; Ch8

; Ch9,10 to first two buttons of Joy1
9            ; Ch9
10           ; Ch10

; last 2 channels omitted: channels ignored

```

```

AssignTab3    ; reads only first 8 channels excepting Ch5, the last 4 channels will be ignored

4            ; Modulation: Futaba PCM 1024

; Ch1 to Ch4 assigned to 4 joy1 analog channels

1            ; Ch1
2            ; Ch2
3            ; Ch3
4            ; Ch4

0            ; Ch5    NOTE that 0 means channel 5 ignored

; Ch6 to Ch8 assigned to 3 Joy2 analog channels

17           ; Ch6
18           ; Ch7
19           ; Ch8

; last 4 channels omitted: channels ignored

TableSelect

; select table 3
3

```

If you need to check what assignments are saved into PIC eeprom you can use the utility **get_assignments.exe** for saving them to *get_assignments.txt* (or the text file you can specify as a command line parameter). Note that *get_assignments.exe* doesn't ask before overwriting an existing destination file to avoid continuous annoying prompts... but if you don't like that refer to section **Computer Sources** below. The format of the output file *get_assignments.txt* is the same of *set_assignments.txt* than is perfectly readable by *set_assignments.exe*. It is also well commented by the program: each table line shows the assignment code of the relative channel and as a comment also the exhaustive explanation of the assignment (channel and the meaning of the assignment code); so you can use it also to verify if you have correctly chosen assignment codes or to get a well commented file to use instead of the original *set_assignments.txt*. Use the version 8axes or 4 axes according to the firmware version you have programmed in the PIC otherwise *get_assignments.txt* could be wrong commented.

Below you can see the **get_assignments.txt** relative to the first example above of *set_assignments.txt* (the one provided in the software package for 8 axes per joystick version); the words in the brackets *normal* or *reversed* (in this example are all normal) reflect the sign of the assignment code (positive / negative) and refer to the way the channel is assigned to the relative joystick control: leaving it as is, or reversing it.

AssignTab1

```

1      ; Modulation: PPM
1      ; Ch1  -> Joy1 X axis (normal)
2      ; Ch2  -> Joy1 Y axis (normal)
3      ; Ch3  -> Joy1 Z axis (normal)
4      ; Ch4  -> Joy1 RotX axis (normal)
5      ; Ch5  -> Joy1 RotY axis (normal)
6      ; Ch6  -> Joy1 RotZ axis (normal)
7      ; Ch7  -> Joy1 Dial axis (normal)
8      ; Ch8  -> Joy1 Slider axis (normal)
9      ; Ch9  -> Joy1 Button 1 (normal)
10     ; Ch10 -> Joy1 Button 2 (normal)
11     ; Ch11 -> Joy1 Button 3 (normal)
12     ; Ch12 -> Joy1 Button 4 (normal)

```

AssignTab2

```

1      ; Modulation: PPM
1      ; Ch1  -> Joy1 X axis (normal)
2      ; Ch2  -> Joy1 Y axis (normal)
3      ; Ch3  -> Joy1 Z axis (normal)
6      ; Ch4  -> Joy1 RotZ axis (normal)
8      ; Ch5  -> Joy1 Slider axis (normal)
4      ; Ch6  -> Joy1 RotX axis (normal)
17     ; Ch7  -> Joy2 X axis (normal)
18     ; Ch8  -> Joy2 Y axis (normal)
0      ; Ch9  -> none
0      ; Ch10 -> none
0      ; Ch11 -> none
0      ; Ch12 -> none

```

AssignTab3

```

1      ; Modulation: PPM
1      ; Ch1  -> Joy1 X axis (normal)
2      ; Ch2  -> Joy1 Y axis (normal)
7      ; Ch3  -> Joy1 Dial axis (normal)
6      ; Ch4  -> Joy1 RotZ axis (normal)
9      ; Ch5  -> Joy1 Button 1 (normal)
10     ; Ch6  -> Joy1 Button 2 (normal)
11     ; Ch7  -> Joy1 Button 3 (normal)
12     ; Ch8  -> Joy1 Button 4 (normal)
0      ; Ch9  -> none
0      ; Ch10 -> none
0      ; Ch11 -> none
0      ; Ch12 -> none

```

AssignTab4

```

0      ; Modulation: ADC mode
1      ; Ch1  -> Joy1 X axis (normal)
2      ; Ch2  -> Joy1 Y axis (normal)

```

```

7      ; Ch3  -> Joy1 Dial axis (normal)
6      ; Ch4  -> Joy1 RotZ axis (normal)
0      ; Ch5  -> none
0      ; Ch6  -> none
0      ; Ch7  -> none
0      ; Ch8  -> none
9      ; Ch9  -> Joy1 Button 1 (normal)
10     ; Ch10 -> Joy1 Button 2 (normal)
11     ; Ch11 -> Joy1 Button 3 (normal)
12     ; Ch12 -> Joy1 Button 4 (normal)

```

AssignTab5

```

1      ; Modulation: PPM
1      ; Ch1  -> Joy1 X axis (normal)
2      ; Ch2  -> Joy1 Y axis (normal)
3      ; Ch3  -> Joy1 Z axis (normal)
4      ; Ch4  -> Joy1 RotX axis (normal)
6      ; Ch5  -> Joy1 RotZ axis (normal)
8      ; Ch6  -> Joy1 Slider axis (normal)
17     ; Ch7  -> Joy2 X axis (normal)
18     ; Ch8  -> Joy2 Y axis (normal)
19     ; Ch9  -> Joy2 Z axis (normal)
20     ; Ch10 -> Joy2 RotX axis (normal)
22     ; Ch11 -> Joy2 RotZ axis (normal)
24     ; Ch12 -> Joy2 Slider axis (normal)

```

AssignTab6

```

0      ; Modulation: ADC mode
0      ; Ch1  -> none
0      ; Ch2  -> none
0      ; Ch3  -> none
0      ; Ch4  -> none
0      ; Ch5  -> none
0      ; Ch6  -> none
0      ; Ch7  -> none
0      ; Ch8  -> none
0      ; Ch9  -> none
0      ; Ch10 -> none
0      ; Ch11 -> none
0      ; Ch12 -> none

```

...

...

...

AssignTab18

```

0      ; Modulation: ADC mode
0      ; Ch1  -> none
0      ; Ch2  -> none
0      ; Ch3  -> none
0      ; Ch4  -> none
0      ; Ch5  -> none
0      ; Ch6  -> none
0      ; Ch7  -> none
0      ; Ch8  -> none
0      ; Ch9  -> none
0      ; Ch10 -> none
0      ; Ch11 -> none
0      ; Ch12 -> none

```

TableSelect

```

1      ; AssignTab1 selected

```

MODULATIONS

ADC mode

Not a modulation really: this mode of operation is used by the firmware when Modulation is set to 0 inside *set_assignments.txt*: instead of reading input signal from *RB0* pin the firmware every time the PIC owns the EP1 or EP2 buffer do an Analog to Digital Conversion of PIC analog input AN0..AN11 in about ½ ms and send the data to PC. The PIC ADC has a resolution of 10 bit (1024 levels) for a voltage range from VSS to VDD (0 to 5V). The firmware first select with the internal switch the analog input AN0 as the input of the ADC, wait an acquisition time of $20 T_{ad} = 26\mu S$ (T_{ad} is the time required to convert 1 bit and is choosed = $1,33\mu S$) for charging the internal 25pF hold capacitor, then start the conversion that lasts $10+1 T_{ad} = 14\mu S$, for a total of about $40\mu S$; than select AN1 and so on for the remaining inputs. The 25pF hold capacitor is charged through an internal resistance of about 3k plus the source resistance R_s that depends on what pots are used; we have choosen the maximum possible acquisition time ($20T_{ad}$) for the best accuracy; however PIC datasheet recommends a source resistance $R_s < 2.5k\Omega$, than pots $< 10k\Omega$ should be used.

PPM (up to 8 proportional channels)

Below you can see the Pulse Position Modulation (PPM) signal timing from Graupner MC-12 (7 channels) measured with oscilloscope:

```
22ms (fixed) |<----->|
400us (fixed) |<>|

->|<---Init.--->|<-Ch.1->|<-Ch.2->|<-Ch.3->|<-Ch.4->|<-Ch.5->|<-Ch.6->|<-Ch.7->|<---Init.--->|<---
+---+      +---+      +---+      +---+      +---+      +---+      +---+      +---+      +---+      +---+
| |      | |      | |      | |      | |      | |      | |      | |      | |      | |
__|8 |_____|1 |_____|2 |_____|3 |_____|4 |_____|5 |_____|6 |_____|7 |_____|8 |_____|1 |__
|<----->| channel pulse (modulated): 1500us for channel = 0%;
                                         1100us/1900us for channel = -100%/+100%;
                                         900us/2100us for channel = -150%/+150%

|<----->| init. pulse (modulated): 22ms minus the duration of 7 channels
                                         11500us for 7 channels = 0%
                                         14300us/8700us for 7 channels = -100%/+100%;
                                         15700us/7300us for 7 channels = -150%/+150%;
```

Since each pulse width is measured between two rising edge, doesn't matter if the radio has a positive ppm signal (as the one above) or negative (level 1 and 0 inverted) and if we use or not an inverting separator stage before *RB0/INT* input pin.

Sanwa/Airtronics PCM1 (8 proportional channels, 10 bits)

Phase1:

_____ - _ - _ - H=====LH=====LH=====LH=====L_
sync1 Ch3 Ch2 Ch1 Ch4 end

Phase 2

---- _ - _ - H=====LH=====LH=====LH=====L-
sync2 Ch5 Ch6 Ch7 Ch8 end

Description: we have two phases, 4 channels each: phase1 is introduced by the synchronization sequence sync1 (4 bit low, 1 high, 3 low, 1 high, 1 low), then 4 channels data (15bits each), then 1 bit low (end); phase2 is introduced by sync2 (4 bit high, 1 low, 3 high, 1 low, 1 high), then 4 channels data, then 1 bit high (end).

Details:

- **bit width:** 0.2ms

- **code pattern:** 1, 2 or 3 bits pulses allowed for data:

- 1 bit pulse = 1
- 2 bit pulse = 10
- 3 bit pulse = 100

examples:

- _____ = 100110
- ----_ = 100110 (same as above)
- _-_-_- = 111111
- ---___ = 100100

- **channel decoding:** every data packet is 15 bits (3 quintets):

- ChX (15bits): H===LH===LH===L
 quintet 3 2 1

- quintets 3, 2 and 1 are decoded into 3 corresponding quartet referring to decoding table below; use the appropriated column depending on quintet; quintet 3 determines 2 bits of data plus per-channel fail-safe infos; quintet values not listed in the table are invalid.

- decoded ChX (12bits, but 2 msbits = 0): H===LH===LH===L
 quartet 3 2 1

decoded ChX is 12 bits but 2 most significant bits are 0 (quartet 3 value can be 0..3 only) so it's really 10bits (1024 points).

Battery Fail-Safe (BFS):

The first 10 bits of phase 1 begins with a low pulse of 4 bits. The following 6 bits indicate if BFS mode is on/off. Here are the possible patterns (Including the first 4-bit pulse):

- 0x233 (1000110011) – BFS Off
- 0x229 (1000101001) – BFS On

Sanwa/Airtronics PCM1 decoding table

<i>quintet value</i>	<i>quartet value for quintets 1 and 2</i>	<i>quartet value for quintet 3</i>	<i>per-channel fail-safe (quintet 3)</i>
0x09	0x2	0x0	no
0x0a	0xa	0x0	no
0x0b	0x8	0x2	no
0x0d	0x6	0x0	yes
0x0e	0xe	0x0	yes
0x0f	0xc	0x2	yes
0x12	0x9	0x3	no
0x13	0x0	0x2	no
0x15	0x5	0x3	yes
0x16	0xd	0x3	yes
0x17	0x4	0x2	yes
0x19	0x3	0x1	no
0x1a	0xb	0x1	no
0x1b	0x1	0x3	no
0x1d	0x7	0x1	yes
0x1e	0xf	0x1	yes

Sanwa/Airtronics PCM2 (6 proportional channels, 9 bits)

Phase1:

```
-----H=====LH=====LH=====L-----
sync1      Ch5      Ch1      Ch2      end
```

Phase 2

```
-----H=====LH=====LH=====L-----
sync2      Ch6      Ch3      Ch4      end
```

Description: we have two phases, 3 channels each: phase1 is introduced by the synchronization sequence sync1 (7 bit high, 1 low), then 3 channels data (20 bits each), then the end sequence (6 bits low, 1 high); phase2 is introduced by sync2 (7 bit low, 1 high), then 3 channels data (20 bits each), then the end sequence (6 bits high, 1 low).

Details:

- **bit width:** 0.3ms

- **code pattern:** 1 or 2 bits pulses allowed for data:

- 1 bit pulse = 1
- 2 bit pulse = 10

examples:

- `--_` = 10110
- `--_--` = 10110 (same as above)
- `---_` = 11111
- `--_--` = 10101

- **channel decoding:** every data packet is 20 bits (5 quartets):

- ChX (20bits): H==LH==LH==LH==LH==L
quartet 5 4 3 2 1

- quartets 5, 4, 3, 2 and 1 are decoded into 5 corresponding duplets referring to decoding table below; for quartet 1 the value to use is the one obtained by clearing the two least significant bits of the quartet (quartet1 & 0x0c); quartet values not listed in the table are invalid.

- decoded ChX (10bits, but msbit = 0): HLHLHLHLHL
duplet 1 2 3 4 5

note that the duplets order in decoded ChX is inverted respect to corresponding quartets order of 20 bits ChX; note also that decoded ChX is 10 bits but the most significant bit is always 0 (looking at the table we can see that duplets relative to (quartet1 & 0x0c) have a maximum value of 1) so it's really 9bits (512 points).

Sanwa/Airtronics PCM2 decoding table

quartet value	duplet value
0x04	0x0
0x05	0x0
0x07	0x2
0x0c	0x1
0x0d	0x1
0x0f	0x3

Futaba PCM 1024 (8 proportional , 10 bits + 2 on/off channels)

frame1:

```
<---sync1---><-----packet1-----><-----packet2-----><-----packet3-----><-----packet4----->
18 bits pulse   Ch1 absolute       Ch3 absolute       Ch5 absolute       Ch7 absolute
+ 000000       Ch2 delta          Ch4 delta          Ch6 delta          Ch8 delta
                                   BFS reset bit          Ch9 bit
```

frame2:

```
<---sync2---><-----packet1-----><-----packet2-----><-----packet3-----><-----packet4----->
18 bits pulse   Ch2 absolute       Ch4 absolute       Ch6 absolute       Ch8 absolute
+ 000011**      Ch1 delta          Ch3 delta          Ch5 delta          Ch7 delta
                                   Ch10 bit
```

Description: we have two frames: frame1 is introduced by sync1 packet: 18 bits pulse + 6 bits low; frame2 by sync2: 18 bits pulse + 8 bits, the first six = 000011 (last2 ignored). Every 2 frames all bits of the frames are inverted; than the firmware does not care of level: it simply toggles bit level after every pulse received, beginning with 0 after sync pulse.

Details:

- **bit width:** 0.15ms
- **6b10b encoding:** every frame packet is composed by 40 bits but once decoded it is only 24 bits long; for every 10 bit we have to apply the 10b6b decoding table below to obtain 6 bits of data (10b values not listed are illegal):

```
<-----encoded packet (40b)----->
<-----10b-----><-----10b-----><-----10b-----><-----10b----->
10b6b      \      \      /      /
decoding    \      /      \      \
              <--6b--><--6b--><--6b--><--6b-->
              <-----decoded packet (24b)----->
```

- **packet format:** each 24 bits decoded packet has the following format:

```
<-----decoded packet (24b)----->
<-2b-><-----4b-----><-----10b-----><-----8b----->
aux      delta          absolute          checksum
```

- **aux bits:** each packet contains 2 aux bits, for a total of 8 aux bits for each frame1 and frame2; depending on the packet and frame they have the following meanings:

```
frame1, packet1 2 aux bits: Frame ID
frame1, packet2 2 aux bits: msb = ?; lsb = Battery Fail Safe (BFS) reset condition
frame1, packet3 2 aux bits: Frame ID
frame1, packet4 2 aux bits: msb = ?; lsb = Ch9 bit
frame2, packet1 2 aux bits: Frame ID
frame2, packet2 2 aux bits: msb = ?; lsb = Ch10 bit
frame2, packet3 2 aux bits: Frame ID
frame2, packet4 2 aux bits: msb = ?; lsb = ?
```

Frame ID specify the frame type:

- **2:** normal frames: almost all frames are normal frames

frame	Frame ID	packet1	packet2	packet3	packet4
frame1	2	ch1 AN ch2 D	ch3 AN ch4 D	ch5 AN ch6 D	ch7 AN ch8 D
frame2	2	ch2 AN ch1 D	ch4 AN ch3 D	ch6 AN ch5 D	ch8 AN ch7 D

(A=absolute N=normal D=delta)

- **0, 1:** failsafe frames: every some ten of seconds a quartet of frames is sent to transmit the failsafe values to use for each proportional channel:

frame	Frame ID	packet1	packet2	packet3	packet4
frame1	0	ch1 AF ch2 D	ch3 AN ch4 D	ch5 AF ch6 D	ch7 AN ch8 D
frame2	0	ch2 AF ch1 D	ch4 AN ch3 D	ch6 AF ch5 D	ch8 AN ch7 D
frame1	1	ch1 AN ch2 D	ch3 AF ch4 D	ch5 AN ch6 D	ch7 AF ch8 D
frame2	1	ch2 AN ch1 D	ch4 AF ch3 D	ch6 AN ch5 D	ch8 AF ch7 D

(A=absolute N=normal F=failsafe D=delta)

the first 2 failsafe frames have Frame ID = 0, the last 2 have Frame ID = 1; inside failsafe frames delta values continue to be sent normally, while absolute values alternate from failsafe and normal depending on channel and frame as described in the scheme above.

BFS (Battery Fail Safe) reset condition can be programmed in quite a few ways: it can be set for low throttle or high throttle; it can also be set to any stick or switch.

- **absolute and delta:** each frame contains absolute 10 bits values of 4 proportional channels and 4 bits delta codes for the remaining 4 proportional channels: for each channel the absolute value is sent in a frame but only 4 bits delta code in the following one for steering servo toward target position with a good approximation given by the formula:

$$\text{channel} = \text{absolute} + \text{delta_value}[\text{delta_code}]$$

For each delta_code we have to find the corresponding delta_value. The table below reports the delta codes sent by TX depending on real delta value (differences from preceding and current absolute values); it is based on infos found on the net and derived by experiments:

real delta value	delta code sent
<= -116	0x0
-115 ... -88	0x1
-87 ... -64	0x2
-63 ... -44	0x3
-43 ... -28	0x4
-27 ... -16	0x5
-15 ... -8	0x6
-7 ... -4	0x7
-3 ... 4	0x8
5 ... 8	0x9
9 ... 16	0xa
17 ... 28	0xb
29 ... 44	0xc
45 ... 64	0xd
65 ... 87	0xe
>= 88	0xf

obviously the receiver cannot go back to the real delta value from the 4 bits only delta code received: the best it could do is to approximate it for minimizing error: this can be achieved by choosing the average value for each real delta value range; anyway this is what the firmware does:

delta_code	delta_value
0x0:	-132
0x1:	-102
0x2:	-76
0x3:	-54
0x4:	-36
0x5:	-22
0x6:	-12
0x7:	-6
0x8:	0
0x9:	6
0xa:	12
0xb:	22
0xc:	36
0xd:	54
0xe:	76
0xf:	102

values for 0x0 and 0xf (very rare anyway) are simply extrapolated from the progression instead of calculated as average value of its range: the relative interval in this case is very large, than in our opinion it is preferable here to risk a greater error than a very large servo overshoot (steer the servo too much over the target).

- **checksum:** it seems that Futaba uses a single bit ECC (Error Correction Code): it allows detection of wrong packets and correction in case of one bit error. How it works: it uses a checksum 8 bits register initialized with 0; the 16 most significant

bits of each packet (bits 23 to 8) are scanned and for each bit = 1 a xor (exclusive or) operation is computed between the value in the checksum register and the number corresponding to that bit according to the list below:

bit23:	0x4a
bit22:	0x25
bit21:	0xa7
bit20:	0xe6
bit19:	0x73
bit18:	0x8c
bit17:	0x46
bit16:	0x23
bit15:	0xa4
bit14:	0x52
bit13:	0x29
bit12:	0xa1
bit11:	0xe5
bit10:	0xc7
bit9:	0xd6
bit8:	0x6b

In other words: $checksum = (bit23 * num_bit23) \text{ xor } \text{ xor } (bit8 * num_bit8)$

The transmitter computes the checksum of 16 msbits of each packet using this formula and send it inside its checksum field. The receiver (and also this firmware) recompute the checksum on the received 16 msbits and do a xor between recomputed value and the received one: if the result is 0 (recomputed and received are identical) it means that very probably there is no error; if the result is one of the values in the list above it means that certainly there is an error but only the corresponding bit is wrong (unless the very rare case of at least 3 bits wrong: infact if only two bits are wrong the result would be the xor between the two relative values but these are chosen to never give any of the other values as a result of a xor between two only values), so we could simply toggle that bit and we are almost certain to have the error corrected; if the result is any other value it means that the packet is certainly wrong and should be discarded because at least 2 bits are wrong and we can't do anything to correct it.

Futaba 10b6b decoding table

<i>10b</i>	<i>6b</i>
0x007	0x3F
0x00C	0x3E
0x00F	0x27
0x018	0x3C
0x01C	0x3D
0x01F	0x26
0x030	0x3A
0x033	0x2B
0x038	0x3B
0x03C	0x30
0x03F	0x0A
0x060	0x38
0x063	0x2A
0x067	0x22
0x070	0x39
0x073	0x21
0x078	0x31
0x07C	0x25
0x07F	0x09
0x0C0	0x34
0x0C3	0x29
0x0C7	0x20
0x0CC	0x28
0x0CF	0x13
0x0E0	0x33
0x0E3	0x23
0x0E7	0x12
0x0F0	0x32
0x0F3	0x11
0x0F8	0x24
0x0FC	0x10
0x0FF	0x08

<i>10b</i>	<i>6b</i>
0x300	0x37
0x303	0x2F
0x307	0x1B
0x30C	0x2E
0x30F	0x0D
0x318	0x2D
0x31C	0x1C
0x31F	0x0C
0x330	0x2C
0x333	0x17
0x338	0x1F
0x33C	0x16
0x33F	0x0B
0x380	0x36
0x383	0x1A
0x387	0x0E
0x38C	0x1E
0x38F	0x06
0x398	0x1D
0x39C	0x15
0x39F	0x07
0x3C0	0x35
0x3C3	0x0F
0x3C7	0x04
0x3CC	0x14
0x3CF	0x05
0x3E0	0x19
0x3E3	0x02
0x3E7	0x03
0x3F0	0x18
0x3F3	0x01
0x3F8	0x00

OS SUPPORT

Windows

Interface developed and tested on Windows XP but should work also on Windows 98 and above. *set_assignments* and *get_assignments* utility are compiled linking them against *LibUSB-Win32* (<http://libusb-win32.sourceforge.net/>); we provide the binaries in the software package; they require *LibUSB-Win32* filter driver (included in the package the version which it has been compiled for).

Linux

We have tested the interface with linux and seems to work perfectly. *set_assignments* and *get_assignments* utilities share the same source with Windows but are compiled linking them against *libusb* (from which *LibUSB-Win32* is derived); we provide the binaries in the software package; they require *libusb* installed in your system.

FreeBSD, NetBSD, OpenBSD, Darwin/MacOSX

As an HID Joystick device RC Joystick NG² should work ok also on these OS but we have not tested it yet. These OS are all supported by *libusb* so the utilities *set_assignments* and *get_assignments* should be smoothly ported on them; but we have not tried it yet so the binaries are not included in the software package.