<u>The USRP under 1.5X Magnifying Lens!</u>

By:
Firas Abbas Hamza

Rev 1.0

Last updated: 12-Jun-2008

This document is dedicated to GNURADO Community (especially the developing team) and is considered as part of GNU Radio project.

Table of Contents

References

The following references have being used to write this document.

- Gnuradio Mailing archives.

- Gnuradio wiki website.

- www.ettus.com

- Dawei Shen tutorials.

- USRP User and Developer's Guide document.

- MSc Thesis "Design of a hardware platform for narrow-band Software Defined Radio applications" by   Kalen Watermeyer

- www.gps-sdr.com

- USRP used Integrated Circuits Datasheets.

Words Must Be Said

- The Gnuradio mailing list is a treasure and some mailing list answers should be gold weighted.

- Please send feedback corrections for any technical mistakes especially those labeled by #FIXME#

- Gnuradio is great and impressive project and the USRP is an amazing device. I always ask my self, how they build this great project? We'll never know the answer.

Starting Notes

1**)** The material of this document was mainly (not all) were taken from the mailing archive posts of (Gnuradio Developers team):
- Eric Blossom
- Matt Ettus
- Johnathan Corgan
- Brian Padalino

2) Some times, I invent the questions when I see important information flipping in the discussion without being asked.

3) To find the original posts in the mailing archive just pick (copy) some ordered words and start searching (paste it) in:
http://lists.gnu.org/archive/html/discuss-gnuradio/
Then you will know who said (or may be asked) these technical information.

4) The answers and technical information in this document are not necessarily 100% correct and error free. Corrections and feedback are highly welcomed and appreciated.

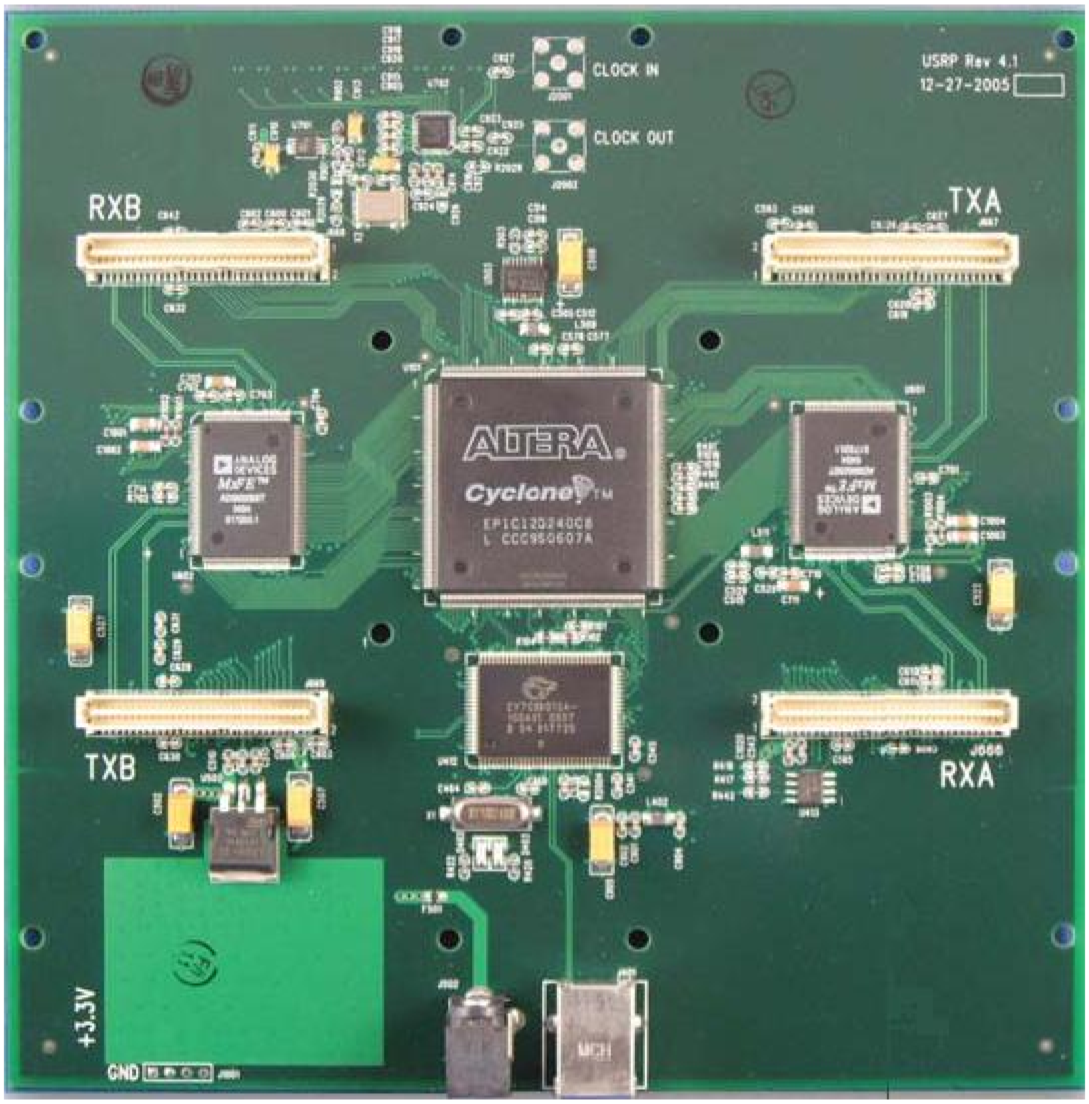
5) I think it is wrong to say that some questions are silly; a big tree was only a small seed in some day.

6) No documentation for inband-signaling and no m-block in this document revision. May be in the future (I hope) it will be included by me or someone else.

7) I hope that; this document is useful.

USRP

The Universal Software Radio Peripheral, or USRP (pronounced "usurp") is designed to allow general purpose computers to function as high bandwidth software radios. In essence, it serves as a digital baseband and IF section of a radio communication system.
The basic design philosophy behind the USRP has been to do all of the waveform-specific processing, like modulation and demodulation, on the host CPU. All of the high-speed general purpose operations like digital up and down conversion, decimation and interpolation are done on the FPGA.
The true value of the USRP is in what it enables engineers and designers to create on a low budget and with a minimum of effort. A large community of developers and users have contributed to a substantial code base and provided many practical applications for the hardware and software. The powerful combination of flexible hardware, open-source software and a community of experienced users make it the ideal platform for your software radio development.



USRP Motherboard

The USRP has 4 high-speed analog to digital converters (ADCs), each at 12 bits per sample, 64MSamples/sec. There are also 4 high-speed digital to analog converters (DACs), each at 14 bits per sample, 128MSamples/sec. These 4 input and 4 output channels are connected to an Altera Cyclone EP1C12 FPGA. The FPGA, in turn, connects to a USB2 interface chip, the Cypress FX2, and on to the computer. The USRP connects to the computer via a high speed USB2 interface only, and will not work with USB1.1.
So in principle, we have 4 input and 4 output channels if we use real sampling. However, we can have more flexibility (and bandwidth) if we use complex (IQ) sampling. Then we have to pair them up, so we get 2 complex inputs and 2 complex outputs.

ADC Section

There are 4 high-speed 12-bit AD converters. The sampling rate is 64M samples per second. In principle, it could digitize a band as wide as 32MHz. The AD converters can bandpass-sample signals of up to about 200MHz. If several decibels of loss is tolerable, then, IF frequency as high as 500 MHz can be digitized. However, if we sample a signal with the IF larger than 32MHz, we will introduce aliasing and actually the band of the signal of interest is mapped to some places between -32MHz and 32MHz. Sometimes this can be useful, for example, we could listen to the FM stations without any RF front end. The higher the frequency of the sampled signal, the more the SNR will be degraded by jitter. 100MHz is the recommended upper limit.
The full range of the ADCs is 2V peak to peak, and the input is 50 ohms differential. This is 40mW, or 16dBm. There is a programmable gain amplifier (PGA) before the ADCs to amplify the input signal to utilize the entire input range of the ADCs, in case the signal is weak. The PGA is up to 20dB. With gain set to zero, full scale inputs are 2 Volts peak-to-peak differential. When set to 20 dB, only .2 V p-p differential input signal is needed to reach full scale. This PGA is software programmable.
If signals are AC-coupled, there is no need to provide DC bias as long as the internal buffer is turned on. It will provide an approximately 2V bias. If signals are DC-couple, a DC bias of VCC/2 (1.65V) should be provided to both the positive and negative inputs, and the internal buffer should be turned off. The ADC VREF provides a clean 1 V reference

DAC Section

At the transmitting path, there are also 4 high-speed 14-bit DA converters. The DAC clock frequency is 128 MS/s, so Nyquist frequency is 64MHz. However, we will probably want to stay below it to make filtering easier. A useful output frequency range is from DC to about 44MHz. The DACs can supply 1V peak to a 50 ohm differential load, or 10mW (10dBm). There is also PGA used after the DAC, providing up to 20dB gain. This PGA is software programmable. The DAC signals (IOUTP_A/IOUTN_A and IOUTP_B/IOUTN_B) are current-output, each varying between 0 and 20 mA. They can be converted into differential voltages with a resistor.

Auxiliary Input/Output Analog Channels:

There are 8 Auxiliary analog input channels connected to low speed 10 bit ADC inputs (labeled AUX_ADC_A1_A, AUX_ADC_B1_A, AUX_ADC_A2_A, AUX_ADC_B2_A, AUX_ADC_A1_B, AUX_ADC_B1_B, AUX_ADC_A2_B, and AUX_ADC_B2_B) which can be read from software. These ADCs can convert up to 1.25MSPS and have a bandwidth of around 200 KHz. These analog channels are useful for sensing the RSSI signal levels, temperatures, bias levels,..etc.

Additionally, there are 8 analog output channels connected low-speed 8bit DAC outputs. These are AUX_DAC_A_A, AUX_DAC_B_A, AUX_DAC_C_A, AUX_DAC_A_B, AUX_DAC_B_B and AUX_DAC_C_B. These DACs can be used for supplying various control voltages such as external variable gain amplifiers control. In addition, there are two additional DACs (labeled AUX_DAC_D_A and AUX_DAC_D_B) which are constructed using 12 bit sigma-delta modulator with external simple low pass filter.

The USRP motherboard connectors (RXA and TXA) share one set of the 4 analog output channels (AUX_DAC_A_A to AUX_DAC_D_A for RXA and TXA) and each have 2 independent analog input channels (AUX_ADC_A1_A and AUX_ADC_B1_A for RXA and AUX_ADC_A2_A and AUX_ADC_B2_A for TXA). The RXB and TXB share their other own independent set. There is also AUX_ADC_REF which can provides a reference level for gain setting if it is necessary.

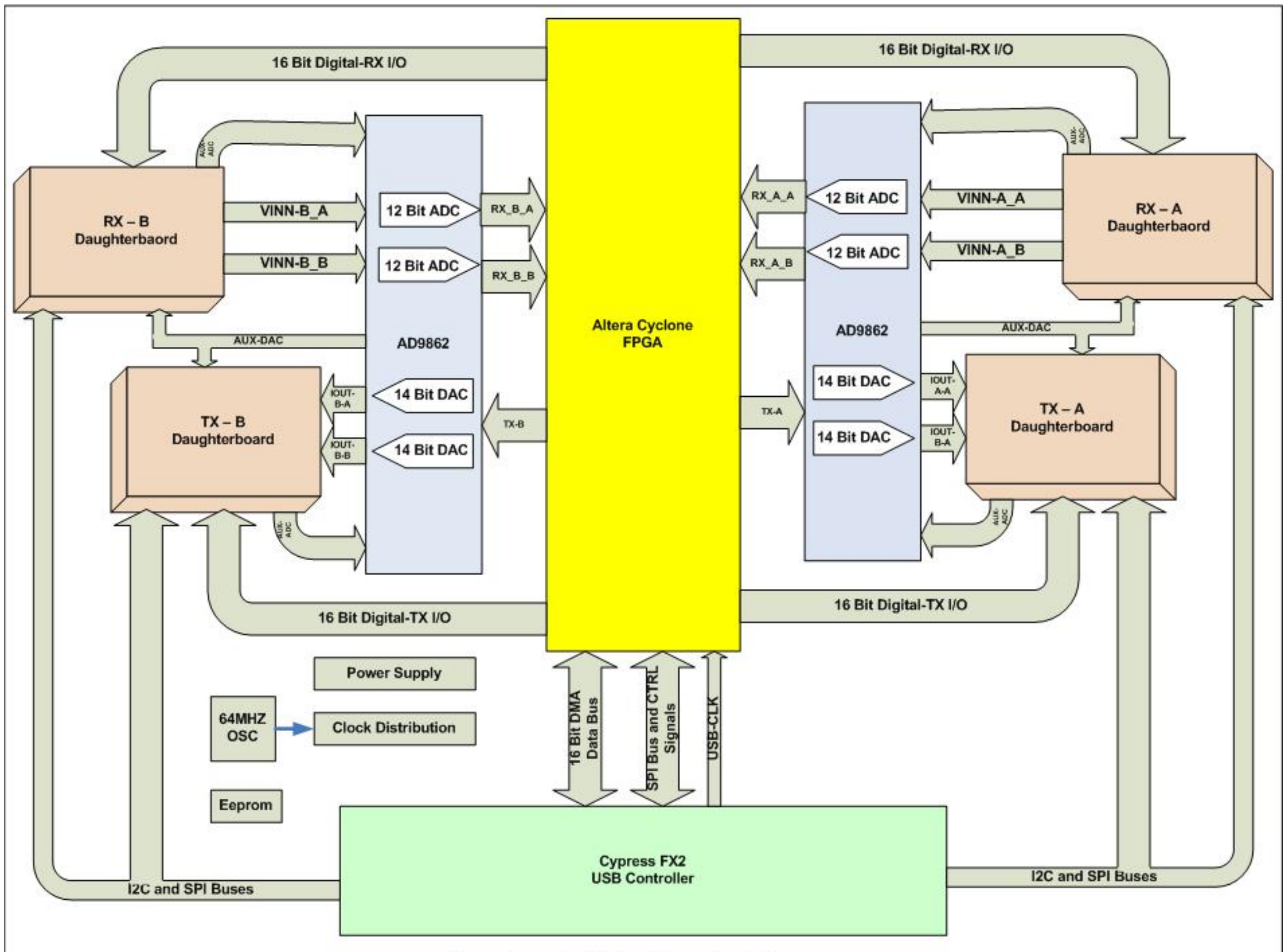
Auxiliary Digital I/O Ports:

The USRP motherboard has high speed 64 bit digital I/O ports. These are divided in two groups (32 bit for IO_RX and 32 bit for IO_TX). These digital I/O pins are connected to the daughterboards interface connecters (RxA, TxA, RxB and TxB). Each of these connectors has 16 bit digital I/O bits. These signals can be controlled from software by reading/writing to special FPGA registers and each can be independently configured either as digital input or digital output. Some of these pins are used to control specific operations on the installed daughterboards such as controlling the selection of receiving RF input port, in automatic transmit/receive mode, controlling power supply feeding for different TX and RX parts, synthesizer lock detection,…etc. It can be also used to implement AGC processing and can be very helpful in debugging FPGA implementations when connected to a logic analyzer.

FPGA

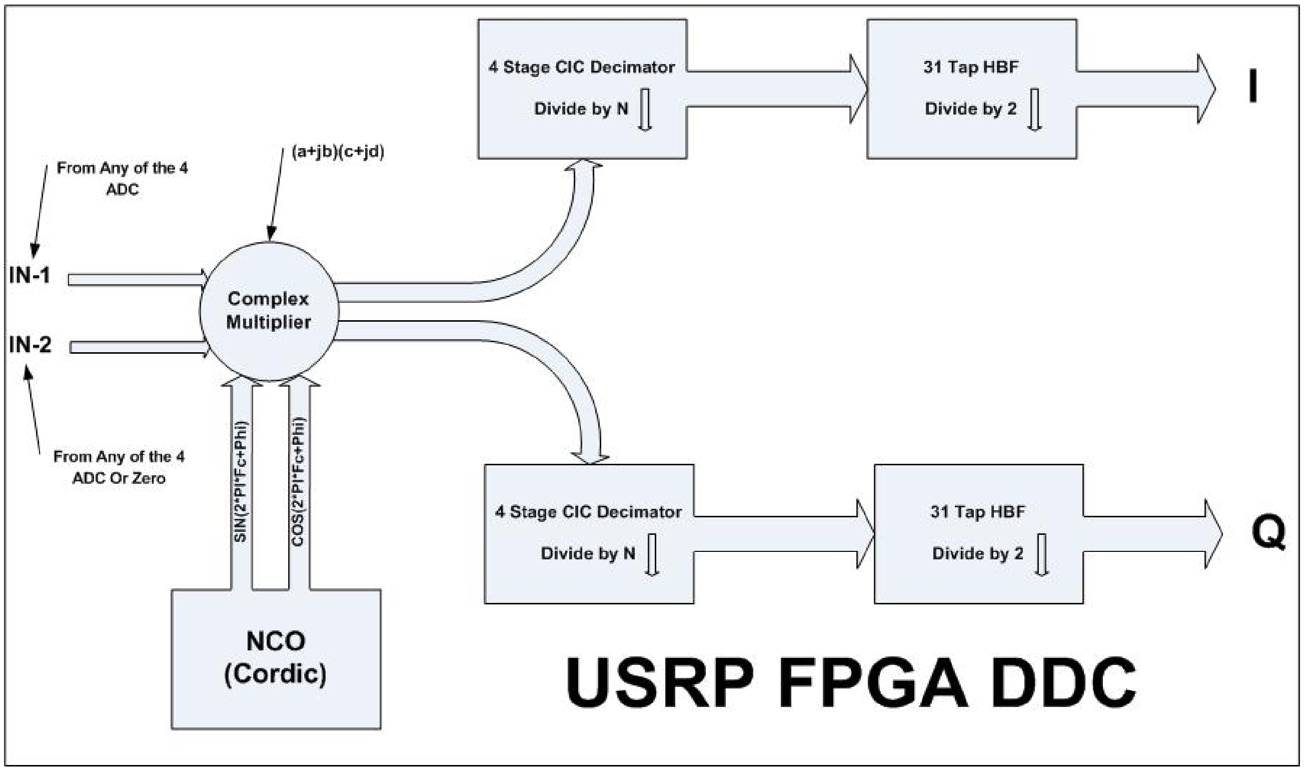
Probably understanding what goes on the USRP FPGA is the most important part for the GNU Radio users. As shown in the figure below, all the ADCs and DACs are connected to the FPGA. This piece of FPGA plays a key role in the USRP system. Basically what it does is to perform high bandwidth math, and to reduce the data rates to something you can squirt over USB2.0. The

FPGA connects to a USB2 interface chip, the Cypress FX2. Everything (FPGA circuitry and USB Microcontroller) is programmable over the USB2 bus.



Simple USRP Block Diagram

The standard FPGA configuration includes digital down converters (DDC) implemented with 4 stages cascaded integrator-comb (CIC) filters. CIC filters are very high-performance filters using only adds and delays. For spectral shaping and out of band signals rejection, there is also 31 tap halfband filters cascaded with the CIC filters to form complete DDC stage. The standard FPGA configuration implements 2 complete digital down converters (DDC). Also there is an image with 4 DDCs but without halfband filters. This allows 1, 2 or 4 separate RX channels.
In the 4 DDC implementation, in the RX path we have 4 ADCs, and 4 DDCs. Each DDC has two inputs I and Q. Each of the 4 ADCs can be routed to either of I or the Q input of any of the 4 DDCs. This allows for having multiple channels selected out of the same ADC sample stream.
The figure below shows the block diagram of the USRP digital down converter.

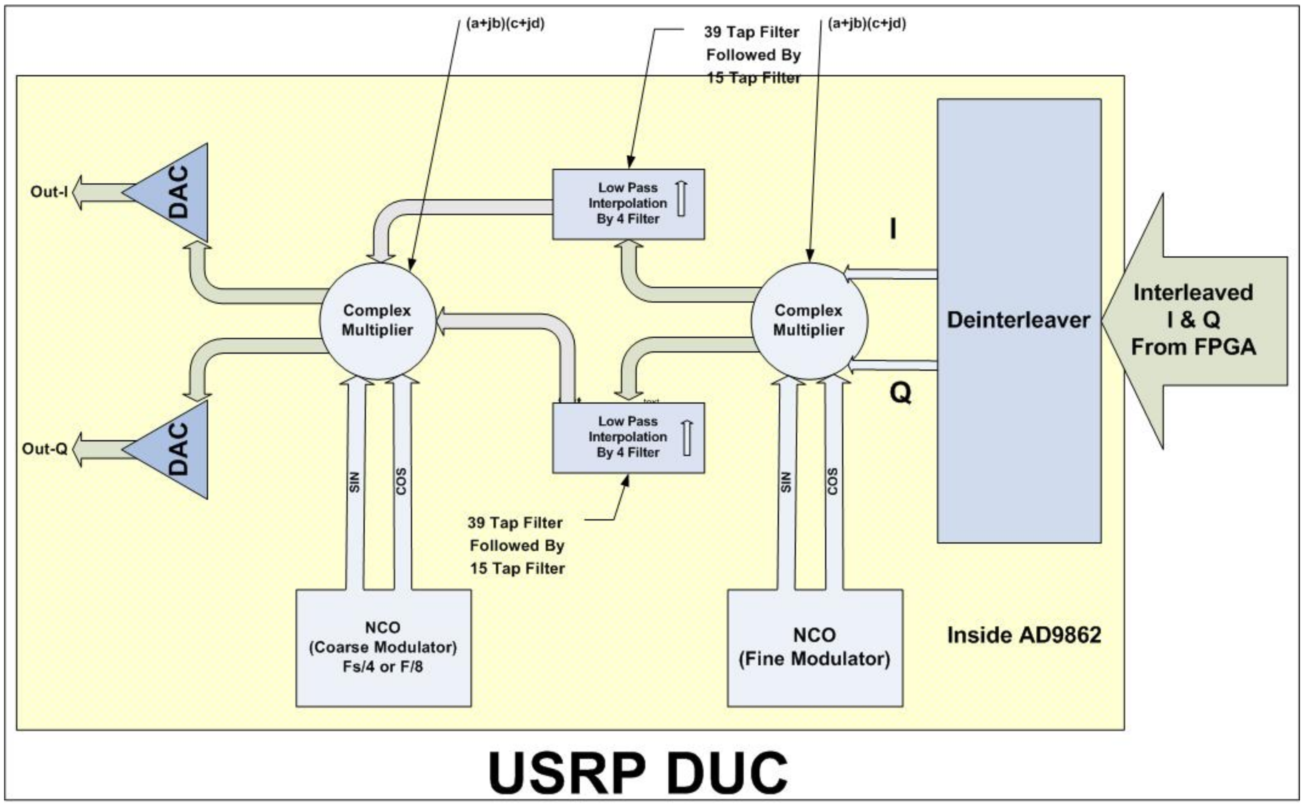


Now let's see the digital down converter (DDC). What does it do? First, it down converts the signal from the IF band to the base band. Second, it decimates the signal so that the data rate can be adapted by the USB 2.0 and is reasonable for the computers' computing capability. The complex input signal (IF) is multiplied by the constant frequency (usually also the IF) exponential signal. The resulting signal is also complex and centered at 0. Then we decimate the signal with a factor N.

The decimator can be treated as a low pass filter followed by a down sampler. Suppose the decimation factor is N. If we look at the digital spectrum, the low pass filter selects out the band [-Fs/N, Fs/N], and then the down sampler de-spread the spectrum from [-Fs, Fs] to [-Fs/N, Fs/N]. So in fact, we have narrowed the bandwidth of the digital signal of interest by a factor of N.
Regarding the bandwidth, we can sustain 32MB/sec across the USB. All samples sent over the USB interface are in 16-bit signed integers in IQ format, i.e. 16-bit I and 16-bit Q data (complex) which means 4 bytes per complex sample. This resulting in a (32MByte per sec/4Byte) 8Mega complex samples/sec across the USB. Since complex processing was used, this provides a maximum effective total spectral bandwidth of about 8MHz by Nyquist criteria. Of course we can select much narrower ranges by changing the decimation rate. For example, suppose we want to design an FM receiver. The bandwidth of a FM station is generally 200 kHz. So we can select the decimation factor to be 250. Then the data rate across the USB is 64MHz / 250 = 256 kHz, which is well suited for the 200 kHz bandwidth without losing any spectral information. The decimation rate must be in [8, 256]. Finally the complex I/Q signal enters the computer via the USB. That's the software world!
Note that when there are multiple channels (up to 4), the channels are interleaved. For example, with 4 channels, the sequence sent over the USB would be I0 Q0 I1 Q1 I2 Q2 I3 Q3 I0 Q0 I1 Q1,

…etc. In multiple RX channels (1,2, or 4) , all input channels must be the same data rate (i.e. same decimation ratio).

At the TX path, the story is pretty much the same, except that it happens reversely. We need to send a baseband I/Q complex signal to the USRP board. The digital up converter (DUC) will interpolate the signal, up convert it to the IF band and finally send it through the DAC.
The digital up converters (DUC) on the transmit side are actually contained in the AD9862 CODEC chips, not in the FPGA (as shown in the figure below). The only transmit signal processing blocks in the FPGA are the CIC interpolators. The interpolator outputs can be routed to any of the 4 CODEC inputs.
In multiple TX channels (1 or 2) all output channels must be the same data rate (i.e. same interpolation ratio). Note that Tx rate may be different from the RX rate.


(a+jb)(c+jd)
39 Tap Filter Followed By 15 Tap Filter
(a+jb)(c+jd)
Out-I
DAC
Low Pass Interpolation By 4 Filter
I
Complex Multiplier
Complex Multiplier
Deinterleaver
Interleaved I & Q From FPGA
Q
Out-Q
DAC
Low Pass Interpolation By 4 Filter
SIN
COS
SIN
COS
39 Tap Filter Followed By 15 Tap Filter
NCO (Coarse Modulator) Fs/4 or F/8
NCO (Fine Modulator)
Inside AD9862
USRP DUC

The USRP can operate in full duplex mode. In this mode, transmit and receive sides are completely independent of one another. The only consideration is that the combined data rate over the bus must be 32 Megabytes per second or less.

Daughterboards

On the mother board there are four slots, where you can plug in up to 2 RX basic daughter boards and 2 TX basic daughter boards or 2 RFX boards. The daughter boards are used to hold the RF receiver interface or tuner and the RF transmitter. There are slots for 2 TX daughter boards, labeled TXA and TXB, and 2 corresponding RX daughter boards, RXA and RXB. Each daughter board slot has access to 2 of the 4 high-speed AD / DA converters (DAC outputs for TX, ADC inputs for RX).

This allows each daughter board which uses real (not IQ) sampling to have 2 independent RF sections, and 2 antennas (4 total for the system). If complex IQ sampling is used, each board can support a single RF section, for a total of 2 for the whole system. Normally, we can see that there are two SMA connectors on each daughter board. We usually use them to connect the input or output signals. No anti-alias or reconstruction filtering is provided on the USRP motherboard. This allows for maximum flexibility in frequency planning for the daughterboards.

Every daughterboard has an I2C EEPROM (24LC024 or 24LC025) onboard which identifies the board to the system. This allows the host software to automatically set up the system properly based on the installed daughterboard. The EEPROM may also store calibration values like DC offsets or IQ imbalances. If this EEPROM is not programmed, a warning message is printed every time USRP software is run.

Each TX daughterboard has a pair of differential analog outputs which are updated at 128 MS/s. The signals (IOUTP_A/IOUTN_A and IOUTP_B/IOUTN_B) are current-output. Also Each RX daughterboard has 2 differential analog inputs (VINP_A/VINN_A and VINP_B/VINN_B) which are sampled at a rate of 64 MS/s.

There are several kinds of daughter boards available now:

Basic TX/RX Daughterboards

Each has two SMA connectors that can be used to connect external up/down tuners or signal generators. We can treat it as an entrance or an exit for the signal without affecting it. Some form of external RF front end is required. The ADC inputs and DAC outputs are directly transformer-coupled to SMA connectors (50Ω impedance) with no mixers, filters, or amplifiers. The BasicTX and BasicRX give direct access to all of the signals on the daughterboard interface (including 16 bits of high-speed digital I/O, SPI and I2C buses, and the low-speed ADCs and DACs). Each of the Basic TX/RX boards has logic analyzer connecters for the 16 general purpose IOs. These pins can be used to help debugging your FPGA design by providing access to internal signals.

Low Frequency TX/RX Daughterboards

The LFTX and LFRX are very similar to the BasicTX and BasicRX, respectively, with 2 main differences. Because the LFTX and LFRX use differential amplifiers instead of transformers, their frequency response extends down to DC. The LFTX and LFRX also have 30 MHz low pass filters for anti-aliasing.

TVRX Daughterboards

This is a receive-only daughter board. It is a complete VHF and UHF receiver system based on a TV tuner module. The RF frequency ranges from 50MHz to 860MHz, with an IF bandwidth of 6MHz.
All tuning and AGC functions can be controlled from software. Typical noise figure is 8 dB. This board is the only USRP daughterboard which is NOT MIMO capable.

DBSRX Daughterboards

Similar to the TVRX board, this is also a receive-only. It is a complete receiver system for 800 MHz to 2.4 GHz with a 3 -5 dB noise figure. The DBSRX features a software controllable channel filter which can be made as narrow as 1 MHz, or as wide as 60 MHz. The DBSRX is MIMO capable, and can power an active antenna via the SMA.

RFX Daughterboards

The RFX family of daughterboards is a complete RF transceiver system. They have Independent local oscillators (RF synthesizers) for both TX and RX which enables a split-frequency operation. Also, it has a built-in T/R switching and signal TX and RX can be on same RF port (connector) or in case of RX only, we can use auxiliary RX port. Most boards have built-in analog RSSI measurement. All boards are fully synchronous design and MIMO capable. For RFX daughterboards RF frequency range, check:
www.ettus.com

Troubleshooting

When first powered up, a LED on the USRP should be flashing at about 3-4 times per second. This indicates that the processor is running, and has put the device in a low power mode. Once firmware has been downloaded to the USRP, the LED will blink at a slower rate.

Power

The USRP is powered by a 6V 4A AC/DC power converter. The converter is capable of 90-260VAC, 50/60 Hz operation, and so should work in any country. If there is a need to use another power supply, the connector is a standard 2.1mm/5.5mm DC power connector. The USRP motherboard itself only needs 5V, but a 6V supply is needed to supply its daughterboards. It draws about 1.6A with 2 daughterboards fixed on it. The power can be checked to be connected to the USRP by seeing a blinking LED on it. If there is no blinking LED, check all power connections, and check for continuity in the power fuse (F501, near the power connector). If the fuse needs replacement, it is size 0603, rating 3 amps.

General Questions

Q) When the USRP is plugged in, the LED to the right of the USB controller blinks at a fast rate of about 3 times per second. Once a USRP application starts, that rate slows down to about 1 time per second. What controls the blinking of the LEDs?

A) The LEDs are controlled by the onboard FX2 USB controller. When you first power the usrp up, a very small program is loaded out of the 256 byte EEPROM. That code puts the AD9862's into a low-power state and runs a tight loop that blinks the led about 3 times per second. See usrp/firmware/src/usrp2/eeprom_init.c. Once the "regular" firmware is loaded, the slow blinking is done in the timer interrupt service routine, isr_tick. See usrp/firmware/src/usrp2/usrp_main.c

Q) I see "O" "U" "u" "a" characters appear on the screen when I run my gnuradio program. I understand they appear when data flow from USRP to PC is stopped or something. I am curious what is the meaning of these characters is? Does faster PC help reducing such data flow troubles?

A) "u" = USRP
"a" = audio (sound card)
"O" = overrun (PC not keeping up with received data from usrp or audio card)
"U" = underrun (PC not providing data quickly enough)

aUaU == audio underrun (not enough samples ready to send to sound card sink)
uUuU == USRP underrun (not enough sample ready to send to USRP sink)
uOuO == USRP overrun (USRP samples dropped because they weren't read in time.

Yes, a faster machine will generally cure this problem. This assumes that you're not asking the USB to do something that it can't. In our best case, we sustain 32MB/s across the USB. I suggest avoiding Intel Celeron or other "cache crippled" parts.

Q) Does the number of 'uO's correspond to the number of samples dropped?

A) No. Overrun detection is currently implemented by polling at approximately 10Hz. If you're trying to receive constantly streaming data, you shouldn't see any uO's.

Q) How the USRP boots (what is the booting sequence)?

A) The USRP itself contains no ROM-based firmware, merely a few bytes that specify the vendor ID (VID), product ID (PID) and revision. When the USRP is plugged in to the USB for the first time, the host-side library sees an un-configured USRP. It can tell it's un-configured by reading the VID, PID and revision. The first thing the library code does is download the 8051 code that defines the behavior of the USB peripheral controller. When this code boots, the USRP simulates a USB disconnect and reconnect. When it reconnects, the host sees a different device: the VID, PID and revision are different. The firmware now running defines the USB endpoints, interfaces and command handlers. One of the commands the USB controller now understands is load the FPGA. The library code, after seeing the USRP reconnect as the new device, goes to the next stage of the boot process and downloads the FPGA configuration bit stream. Once the firmware has been downloaded to the FX2, it sets an internal register and reboots itself, this time presenting custom product and vendor ID's, which the host detects as the FX2 disconnecting followed by the connection of a custom USB device.

Q) What tools are you using to do the FPGA programming? And which tools did you use to layout the USRP motherboard?

A) The FPGA is programmed in Verilog, and compiled with Quartus from Altera. The motherboard layout was done in PADS, but the layout files are not distributed. The schematics are done in gEDA.

Q) Can we use DC-DC converter with outputs 5V to power USRP?

A) The USRP motherboard can operate from 5V, but most of the daughterboards regulate the 6V down to 5V, so we really need about 5.5 to 5.75 V to operate correctly.

Q) What is the maximum DC supply voltage for the USRP? Can we use 13.8V?

A) No, it isn't. The USRP itself uses a linear regulator, so putting 13.8 V in will result in more than 10 additional watts being dissipated. Also, many of the capacitors on the daughterboards are sized for 6V input, and might blow with more than 10 V.

Q) What is USRP motherboard fuse rating?

A) The fuse size is 0603. Rating is 3Amps.

Q) I think the EP1C12Q240C8 FPGA runs at much higher clock speeds than 64MHz.

A) When the FPGA is routed, there isn't much more slack above the 64MHz for Fmax (or so I thought).

Q) How control signaling is done? What is the protocol for it? When I use these lines:

```
    src = usrp.source_c (0, decim)
    src.set_rx_freq (0, IF_freq)
    src.set_pga(0,20)
```

How actually it gets translated and conveyed to the FPGA master_control module on FPGA to do the needful?

A) Please take a look at the code in usrp_standard.{h,cc}, usrp_basic.{h,cc}, and usrp_prims.{h,cc}. They all go over the USB as control messages to endpoint 0. The control messages are parsed in the FX2 and acted upon. Generally speaking they end up issuing transactions on the I2C bus or SPI bus.
The FPGA registers, AD9862 registers and daughterboards appear on the SPI bus. The configuration EEPROM's and daughterboards appear on the I2C bus. See the USRP schematics for details.
The usrp/firmware/include/usrp_spi_defs.h contains the definition for the SPI bus. The usrp/firmware/include/usrp_i2c_addr.h contains the standard I2C addresses. Some daughterboards use others that are a function of which slot the daughterboard is plugged into.

Q) How big are the USRP buffers?

A) The USRP has buffers at three points in the path:
On the host: currently 8MB TX, 8MB Rx. (#FIXME#)
On the Cypress FX2 USB peripheral: 2KB TX and 2KB Rx. (#FIXME#)
On the FPGA: 4KB TX and 4KB RX.

Q) When I have two RX Channels, then how exactly is the rx_buffer used to store and keep the data coming from 2 RX paths separated?

A) The different paths alternate in the FIFO. The data is in this order:

CH1-I
CH1-Q
CH2-I
CH2-Q
etc.

Q) I wish to build just one RX with halfband filter and one TX. The present usrp_std.vh has 2rx, 1tx or 4rx. What do I need to do to?

A) Edit the file config.vh , then uncommented the line:

```
//`include "../include/common_config_1rxhb_1tx.vh"
```

And comment the line:

```
`include "../include/common_config_2rxhb_2tx.vh"
```

By using “//”, this will now includes 1 TX and 1 RX with halfband filter.

Q) Things which could be controlled from Python on FPGA and AD/DA converters are: Gains and Decimation Rate. Can bits/sample of AD/DA converters be controlled from python? What else is controllable from Python?

A) The ADCs always sample at 12 bits. The DACs are always 14 bits. The data processed in the USRP is always 16 bits. From python you can switch to 8-bit samples over the USB. This allows you to double your sample rate make it (16MSPS).

Q) I've been playing with the usrper application. I am able to read and write to the 9862 chips okay but all attempts at using i2c_read fail? What is the memory map?

A) I usually prefer to do this kind of experimentation from Python, but in any event you're probably using an invalid I2C address. See usrp/firmware/include/usrp_i2c_addr.h
Note also, that if what you're really trying to do is read the contents of one of the EEPROM's, you're much better off using read_eeprom.

>From python:

```
  u = usrp.sink_c(0)        # or u = usrp.source_c(0)

  # returns a string
  s = u.read_eeprom(i2c_addr, eeprom_offset, nbytes)
  print s

  # returns a string
  s = u.read_i2c(i2c_addr, nbytes)
  print s
```

Q) How we control auto transmit/receive switch delay?

A) This allows someone (from Python) to manually adjust the timing offset of the auto-transmit/receive switching to better align with the transmitted data when using an external transmit/receive switch.
In a previous code, the auto transmit/receive function would switch based on whether or not there was data in the transmit FIFO of the FPGA. Unfortunately, this would not take into account the pipeline delay of the FPGA interpolator and AD9862 DAC up converter. Depending on the interpolation ratio used in the transmit side, this could result in the ATR switching signal going high up to 25 us early, and going low up to 35 us before the data was completely transmitted. If someone were using a fast external Rx/Tx switch, this could result in slicing off the end of a

transmission.
The new capability adds an independent, configurable delay to the rising and falling edges of the ATR signal, measured in clock ticks. To use, once one has already configured ATR operation as needed, one needs to call two new daughterboard functions:

```
subdev.atr_set_tx_delay(clock_ticks_to_delay_tx) # Rising edge
subdev.atr_set_rx_delay(clock_ticks_to_delay_rx) # Falling edge
```

that would be 'set_atr_tx_delay' and 'set_atr_rx_delay'.

...where 'subdev' is the object created for the daughterboard in use.

Unfortunately, the best way (right now) is to figure out what values to poke in here are to measure the offset on an oscilloscope between the onset of the ATR signal and the power output from the daughterboard.
We're working on a way to ask the FPGA itself what values should be set (or even set these by default), but haven't worked out yet the best way to do this. Different external Rx/Tx switches have different needs so there isn't a one size fits all.
Empirically measured, though, the pipeline delay through the USRP transmit path is about 50 ticks + 3 ticks * interpolation rate. It seems the falling edge needs somewhat longer than this to completely drop to zero energy and this may be data dependent, so again, the best course is to measure with your particular application.

Q) What is the USRP Clock reference instability?

A) The clock has a 20 ppm specification, but at room temperature, It is typically within 5 ppm.

Q) How the expected in-band TX timestamps may work?

A) Here is the order I was thinking:
  - FX2 receives packet to be sent to FPGA
  - FX2 sends entire 512 byte packet to the FPGA
  - FPGA has state machine to process the packet which does the following
    - Check to be sure it can figure out where the packet has to go
    - Check to be sure the FIFO is not full
    - Read the length of the packet
    - Fill up the receiving FIFO until it is full or total length of packet has been reached
    - If the packet length was reached (which might be less than 512), skip the rest since it's padding
    - Else if the FIFO is full, loop and fill the FIFO until the packet length has been reached
  - FPGA has command and channel state machines which do the following with the tx_cmd and tx_channel FIFOs
    - Read out timestamp and length from packet
    - Ensure timestamp is in the future
    - If timestamp is in the past, skip the rest of the packet and set

appropriate flag
  - Else wait until the current time is reached
    - If current time is reached, send out the size of the packet

Q) I was just wondering whether the USRP has got some temperature specs that I should take into account.

A) All of the chips in the USRP are specified over at least the 0-70C range. At room temperature the USRP can run all-out continuously for days at a time. Any enclosure should take the power consumption of the system into account. This can be as high as 15-18 Watts depending on what daughterboards you use.

Q) Is the loopback interface working on the USRP, and how to enable it?

A) There is a loopback interface in the current FPGA code. It's enabled by writing to the FR_MODE register. Grep the verilog for "loopback", and look at fpga_regs_common.{h,v}

Q) I was wondering why can the USRP "only" achieve 32 MBytes/sec, i.e. 256 Mbits/sec whereas the USB 2.0 specifications are 480 MBits/sec?

A) The USRP easily does 256 megabits per second (32 megabytes/s). The USB 2.0 raw signaling rate is 480 megabits per second, or 60 megabytes per second. You can't get the full 480 because there is overhead from packet headers, time between packets, etc. We could probably squeeze a little more bandwidth out of the bus, but it isn't a priority for now. The USRP needs data to go both ways, so there are some firmware delays in the 8-bit processor that runs in the interface chip. If someone cared, it is probably possible to reprogram the firmware so that when no data is being transmitted, the automatic hardware mode is used. The firmware would have to know, or be told, when to switch in and out of this mode. But this would gain 25% more bandwidth!

Q) The USRP RFX boards have about 20 MHz bandwidth on both TX and RX sides, and can operate full duplex as long as you separate the TX and RX frequencies (TX and RX have separate PLL synthesizers). Is there a way to move 20MHz-wide signals between the host machine and the RFX?

A) 20 MHz bandwidth indicates the width of the baseband filters. Thus, you can tune digitally anywhere +/- 10 MHz from the LO. If you use 16 bit samples, you can't get that much over the USB bus at once.
So why is 20 MHz useful? Here are 3 example scenarios:
- You could simultaneously receive or transmit 2 separate signals which are 4 MHz wide each, which are separated by 12 MHz.

- You can use 8 bit samples, and get 16 MHz of bandwidth. If you were a radio astronomer, you could use 4-bit samples and get 32 MHz of bandwidth.

- You could do a wideband modulation and demodulation in the FPGA. You would be able to deal with much wider signals, and send a lower bit rate over the bus.

Q) The DC offset is automatically removed in hardware, but we can turn this feature off. Is there a software flag in the code I can switch it off?

A) See:
http://gnuradio.org/trac/wiki/UsrpFPGA

Under "Common Registers":
15 FR_DC_OFFSET_CL_EN DC offset control loop enable

FPGA Verilog Questions

Q) Would be it possible to modify the USRP FPGA Verilog code?

A) Yes. It is possible. The top level of the verilog source for the FPGA can be found here:
usrp/fpga/toplevel/usrp_std/usrp_std.v
the bulk of the code is here:
usrp/fpga/sdr_lib/*.v

Q) I would like to recompile all the verilog files for the FPGA in order to have a better understanding of the role of each module. I believe the program used to compile the files was Quartus II. Is that correct?

A) If you want to build the FPGA .rbf file from source (not required; we provide pre-compiled .rbf files in usrp/fpga/rbf directory), you'll need Altera's no cost Quartus II development tools. We're currently building with Quartus II Web Edition.
The project file is usrp/fpga/toplevel/usrp_std/usrp_std.qpf. The top-level verilog file is usrp/fpga/toplevel/usrp_std/usrp_std.v. The bulk of the verilog modules are contained in usrp/fpga/sdr_lib/*.v

Q) How the data coming/going to the AD9862 is handled in the FPGA?

A) In the FPGA RX Side, in the DC offset removal process, they are assigned on input. Here is where it happens:
http://gnuradio.org/trac/browser/gnuradio/trunk/usrp/fpga/sdr_lib/adc_interface.v#L34
we can see that the input of the rx_dcoffset module is assigned:
.adc_in({adc0[11],adc0,3'b0})
Which is a 1 bit sign extension, the 12 original bits, then 3 bits of 0's.

While in the FPGA TX Side (as can be seen bellow):
http://gnuradio.org/trac/browser/gnuradio/trunk/usrp/fpga/toplevel/usrp_std/usrp_std.v#L179

whatever 16-bits are coming from the tx_chain module is truncated by 2-bits and sent out to the DACs.

Q) The tx_chain.v module uses a module called phase_acc. It takes as inputs (among others) a 7bit serial address and 32 bit serial data and outputs the 32bit phase. Does anyone have any idea what that module does and what its purpose is in the tx_chain?

A) It is not enabled. If enabled, this code would implement digital up conversion in the FPGA, instead of using the DUC in the AD9862. N.B., this code hasn't been tested in something like a couple of years and may have suffered bit rot.

Q) I was looking at the serial_io.v module but I am not sure what its purpose is or what it does?

A) The serial_io.v is only used for writing and reading the FPGA registers.

Q) The data arrives as a serial input (from USB bus) to the FX2 chip. Then, it gets to the FPGA as a 16bit parallel input from the GPIF bus. That input is called usb_data in the FPGA. However, upon reading the verilog code of the serial_io.v module, it seems that this module does a serial to parallel conversion of the SDI input and puts it into the 32 bit serial_data output register. I am confused as to whether the serial data from the USB is converted to parallel in the FX2 chip or whether the data sent to the FPGA is serial but a verilog module inside the FPGA converts it to parallel. So I am not sure whether the data coming from the USB to the usrp_std.v module is the 16bit input usb_data or the SDI input.

A) The actual data to/from the DACs/ADCs (DUCs/DDCs) goes across the 16-bit GPIF bus.(#FIXME# not all the asked question is answered)

Q) I am looking at the toplevel module ustp_std.v It is my understanding that the data to be sent coming from the PC or the data received going to the PC is the "usbdata" inout. The other inouts io_tx_a, io_tx_b, io_rx_a and io_rx_b are the signal going to/coming from the daughterboards. I am still confused on what is the purpose of the following inputs: rx_a_a, rx_b_a, rx_a_b, rx_b_b and the following outputs: tx_a, tx_b?

A) Take look at AD9862 Data Sheet!!!!!!!!!!

Q) I am looking at the transmission buffer (tx_buffer.v) verilog code. So it seems that it uses two clock signals. The txclk signal is used to read from the fifo_4k while the usbclk signal is used to write to the fifo_4k. What are the clock speeds for those clock signals?(#FIXME# is the question correct?)

A) The USB clock is 48 MHz, provided by the FX2.(#FIXME# is the answer enough?)

Q) One of the inputs to the usrp_std module is SDI. What is the origin of this input?

A) The 8051 bit-bangs the SDI data, clock, and strobe lines. They are "slow" compared to other FPGA actions. See usrp-*/firmware/src/usrp1/spi.c

Q) The memory allocated for each of the FIFO in tx_buffer and rx_buffer is twice their capacity.....i.e 65K 'bits'(8192 bytes) as seen from the compilation report of Quartus. Is there any specific reason?

A) The FIFOs are 4K lines long, with each line 16 bits.

Q) In the cordic.v, 16 bit vector xi, yi and zi are passed as input. Then cordic constants are pre-defined. What "wire [bitwidth+1:0] xi_ext = {{2{xi[bitwidth-1]}},xi};" statement is doing. Is it just normal concatenation and assignment?

A) Yes, we are sign-extending xi.

Q) The @posedge clock, if reset is set, then x0, y0, z0 is initialized to '0'. Next, when enable is set, z0 is assigned all bits except the first two. And the first 2 bits are used for the case statement. In this case statements, the first 2 bits are used to determine the quadrants and accordingly adjust the quadrant (angle) for xi_ext and yi_ext by 90 or 180 or 270 or 360 degree. But I am not sure. Is this correct?

A) Yes

Q) Many instances of cordic_stage are defined and they are passed input values x0, y0, z0 and constants c00 etc. I am not sure what "#(bitwidth+2,zwidth-1,0)" in this instance definition does?

A) The “#( )” is used to pass parameters to the module being instantiated.

Q) In cordic_stage.v, if reset is set then xo, y0 and zo are set to 0. And when enable is set, value of z is checked. My understanding of cordic logic is that, if say that current angle is 45, and desired is 20, then subtract from the current angle, and make corresponding adjustments on x axis and y axis variables. This statement checks if "z_is_pos" and does next operation respectively. I am not sure what "xi - {{shift+1{yi[bitwidth-1]}},yi[bitwidth-2:shift]}" is actually doing. My understanding of cordic is that based on angle adjustments, operations of multiples of 2 are carried out on x-axis and y-axis variable. And {} statement are used for concatenation in verilog.

A) Sign extension

Q) In cordic.v, xo, yo and zo are in continuous assignment mode and are updated every time a x12, y12 or z12 value changes respectively. Not exactly sure how the exact desired angle is obtained in cordic.v. The algorithm that I read made continuous adjustments on 'current angle' so that it converged to the desired angle. I don't see that step happening in cordic.v?

A) Here how it goes. If the angle is positive, it is reduced. If the angle is negative, it is increased. It always gets closer to zero.

Q) How we write a test bench for tx_chain.v?

A) To write a test bench, you would need to set the interpolation frequency properly and if you wanted to use the CORDIC, that would have to be setup properly as well for the phase accumulator. You would then strobe your samples in at a rate of Fs where Fs is some fraction of the total clock speed and Fs*interp_rate = the clock rate. You then have to strobe every Fs to send your samples through the chain.
You also need to assert the interpolator_strobe and sample_strobe on a consistent frequency basis - eg: once every 15 clock cycles. You can't just randomly assert and de-asserts them. You have no frequency set. I believe this frequency should be the phase difference between samples of the CORDIC for each sample that goes in where 2^31-1 is equal to 2*PI. You probably want to see a sine wave or some filtered signal - try a pulse train, {1, 0, 1, 0, ...} or you can download a math_real.v module I wrote from here:

http://www.gnuradio.org/trac/browser/gnuradio/branches/developers/zhuochen/simulations/burst_test/math_real.v
I used it in the test bench here to create some sines and cosines:

http://gnuradio.org/trac/browser/gnuradio/branches/developers/zhuochen/simulations/burst_test/test_chan_fifo_reader.v

Q) Do people just burn their Verilog code to the board to test it?

A) For pre-synthesis, RTL simulation and testing, I use ICARUS Verilog under Linux and GTKWave. Note that if you plan on using it with ICARUS Verilog, you will need one of the latest developer snapshots as the code I wrote found some problems in their real value handling. It is definitely NOT recommended you just put it into an FPGA and run with it as you really have no idea what's going on inside there. Accurate modeling can really help with your debugging time.

Q) I went through the mailing list and figured out that the current Verilog/VHDL code implementation occupies 95% of FPGA's resources. However, there were some mails that pointed that reducing some receiver functionalities could free some FPGA resources. How?

A) There's a header file:
GNU_Radio\usrp\fpga\toplevel\usrp_std\config.vh
That header controls the build configuration and is now functional. Modify it to use the file: GNU_Radio\usrp\fpga\toplevel\include\common_config_1rxhb_1tx.vh
This is how:

```
// Uncomment this for 1 RX channel (w/ halfband) & 1 transmit channel
`include "../include/common_config_1rxhb_1tx.vh"
```

This will free up a lot of space on the FPGA for experimentation!

Q) In rx_chain.v, what are the following signals for?
 -sample_strobe.v
 -decimator_strobe.v
 -hb_strobe.v
 -serial_addr,serial_data,serial_strobe
 -debugdata,debugctrl

A) sample_strobe follows the full speed samples coming from the ADC and feeds the input of the decimating CIC filter.

decimator_strobe is generated by the decimating CIC filter to feed the samples into the halfband FIR filter at the specified and decimated rate. When the decimation by N takes place, you get a sample 1/N cycles. This ratio is preserved throughout that part of the chain.
The hb_strobe comes out of the halfband decimating FIR filter. This is a constant decimation by 2, but is asserted when the FIR filter has an output to pass out of the RX chain.

serial_* is the interface used for register writing. If you want to modify any of the registers within the module, the simple serial interface from the FX2 -> FPGA is what changes them. In this case, they are used to modify the register for phase accumulation within the FPGA for the cordic.

debug* is just used for debug as the name suggests. In the current implementation in the trunk, it looks like the input to the halfband FIR filter is what output on the debug bus. (#FIXME#)
For informational purposes, the interpolation strobes and filtering modules are found here:
http://gnuradio.org/trac/browser/gnuradio/trunk/usrp/**fpga**/sdr_lib/master_control.v
http://gnuradio.org/trac/browser/gnuradio/trunk/usrp/**fpga**/sdr_lib/cic_interp.v
http://gnuradio.org/trac/browser/gnuradio/trunk/usrp/**fpga**/sdr_lib/strobe_gen.v

The register is a simple register updated once the serial data is properly written (read: atomic operation). The strobe_gen is a down counter which resets to the input of whatever that rate register is feeding the module when the counter gets down to 0.

Q) How many clock cycles does it take the cordic module to output valid data and why is the cordic algorithm implemented (in cordic.v) fixed at 12 iterations?

A) It is a pipelined module which takes samples at the full rate and through 12 pipeline stages outputs the rotated vector. Why 12 was picked is probably a size issue since you create registers and flops for each stage. You can make the pipeline deeper and even variable if you want. There is a TODO in there which suggests making it a variable length - not sure if the comment means at runtime or at build time, but have a go at it.

Q) Can we use Mega cell Altera blocks?

A) Yes. For example, like the tx_usb_fifo and rx_usb_fifo, you can just tell Quartus to generate the appropriate dual clock FIFO "megacell" for you. In this way, Matt did that to generate the fifo_2k and fifo_4k blocks that we're currently using. See the Quartus manual and the "Cyclone Device Handbook" for supported configurations.

Q) The following is the setting_reg module
```
module setting_reg
( input clock, input reset, input strobe, input wire [6:0] addr,
    input wire [31:0] in, output reg [31:0] out, output reg changed);
```
In the :
..\gnuradio-3.0.2\usrp\fpga\sdr_lib/master_control.v module,
I see settting_reg module is called as follow:
```
setting_reg #(`FR_MASTER_CTRL)
sr_mstr_ctrl(.clock(master_clk),.reset(1'b0),.strobe(serial_strobe),.addr(serial_addr),.in(serial_dat
a),.out(master_controls));
```
However, in:
..\gnuradio-3.0.2\usrp\fpga\sdr_lib/setting_reg.v module, the setting_reg module has an output signal called "changed".Can anyone please tell me what will happen to that "changed" signal?

A) The “changed” signal is not used in most places, and is left out.

AD9862 Codec Questions

Q) I was looking at the AD9862 data sheet, and it seems that in the transmit path, there is only one 14bit input but the chip outputs 2 TX signals. Does this mean that if we want to output two signals, the data has to be interleaved before being sent to the AD chip? I am not sure why there are 2 outputs but only 1 input?

A) The data sent to the AD9862 is interleaved, IQIQIQIQIQ.

Q) I am trying to figure out how the AD9862 MxFE is setup:
RX Side:
1. Is the internal DLL used to sample faster than 64MSPS? If so, what is the DLL rate? Is it possible this would ever want to be used?
2. Is the Hilbert Filter ever used?

TX Side:
1. Is the Hilbert Filter ever used?
2. Is the NCO ever used?
3. Is the interpolation filter ever used?
4. Is the fixed [Fs/4,Fs/8] selectable mixer ever used?
5. Is the TX data ever sent as real-only instead of I/Q from the FPGA when it is actually a complex signal?

Aux Side:
Are any of the auxiliary ADC/DACs used for AGC/VCO setting?

A) RX Side:
1- The DLL is used to double the sample rate so that the *DACs* can sample at 128MHz. The ADCs stay at 64MHz.
2- No

TX Side:
1- No
2- Yes, all the time (both coarse and fine). In the up converter (in the AD9862, from figure 3 in the 9862 datasheet), they split the complex multiply into 2 parts -- coarse and fine. The fine part (block D) runs at 1/4 of the sample rate. This means that it can only move the freqency 1/4th as far. The block is then followed by the 4x interpolation (Block C), and then the coarse modulation, Block B. Block B only moves the signal +/- fs/4 or fs/8.
3- Yes, always.
4- Yes
5- No, we always send I/Q to the 9862. There are a couple of use cases, where you might want to send real data, but we don't implement them.

Aux:

Yes, all of them. Details depend on the specific daughterboard. The aux DAC/ADC's are all run to the daughterboards.

Q) How AD9862 registers maintained?

A) The AD9862 is controlled over SPI. The code that sets up the AD9862's is contained in usrp_basic.cc and usrp_standard.cc. Most of it is in the constructors. You may also want to take a look at the USRP motherboard schematic to see how everything is wired together.

Q) I have two Basic_Tx daughterboards, so each one uses different AD9862. Are the 2 AD9862 both get there clock in from the usrp motherboard, and that is the same clock?

A) The AD9862s (which are on the motherboard) both get the same clock from AD9513 clock distribution IC. The only difference is that the clock signal goes through separate (but identical) filters on its way to each AD9862.

Q) Would you please iterate more on the concept of AD9862 PGA?

A) It's an analog amplifier with a software controllable gain. On the Rx path it has steps of 1.0 dB before ADC. On the TX path, the steps are 0.1 dB after the DAC. See the AD9862 datasheet for details. The PGA is controlled over the SPI serial bus from the FX2 (as are all the AD9862 registers).

Q) I need a way to update the AUX_DAC port on the 9862, but apparently that is handled by the USB microcontroller! Is that correct? To set up the AUX_DAC do I have to waste USB bandwidth?

A) The USRP is wired such that only the FX2 can control the AD9862s and you cannot change it from FPGA. However, you can change this hardware wiring by adding R2001, R2004, and R2005 to the USRP. All should be Zero-ohm resistors. Be careful, because you will now have multiple drivers of these nets. You'll need to have the FX2 STOP driving them before you make the FPGA drive them.
This will allow the FPGA to control the AD9862s. You can also make the FPGA control the tuning of the RFX-series daughterboards.

Controlling FPGA Registers Questions

Q) Can anyone tell me where the registers are and how I can read/write to them?

A) Once you create a usrp.source_c() or usrp.sink_c() object, you can call:

```
u = usrp.source_c()
u._write_fpga_reg(regno, val)
```

The FPGA registers are write-only. You may call:

```
u._read_fpga_reg(regno)
```

...but the return value will either be zero or a semi-documented set of debugging values unrelated to the register number you put.
The FPGA register definitions are in:
usrp/firmware/include/fpga_regs_common.h
usrp/firmware/include/fpga_regs_standard.h

NOTE: It is possible to damage your hardware by incorrectly setting these registers I believe setting_reg.v is instantiated wherever there is a register which can be written to. Being able to have access to every register makes for a very hairy problem. The FPGA registers are not set using I2C, they are set with SPI. There is already a mechanism for reading back values over SPI, but it is only set up to read back 8 values now, mostly for reading I/O pins and RSSI levels, but not every register. See serial_io.v and setting_reg.v

Q) What FPGA registers are used for SPI, I2C control? Or are they just a standard registers?

A) All the I2C and SPI stuff is outside the FPGA. The SPI is a serial bus which is used to set registers on the daughterboards, the FPGA, and the AD9862 are controlled from the FX2 only.

Q) I'm trying to read and write the FPGA registers using just the libusrp. My write returns "true", but the read returns 0, regardless of which FPGA register I read/write to. For example:

```
if (utx->_write_fpga_reg(FR_ADC_OFFSET_0,0x7777))
int readregvalue2 = (utx->_read_fpga_reg(FR_ADC_OFFSET_0));
```
Why is that?

A) The FPGA registers are write-only. When reading, instead of the register contents, you get 'readback' register contents which are actually a number of useful signals inside the FPGA. See line numbers 304 and 305 of usrp_std.v to see what signals are read back. The typical way of dealing with write-only registers is to maintain a shadow copy in your code that you update every time you write to a register, and then if you later want to "read" the register, just look at the shadow copy. You can also use this if you want to do a read-modify-write.

Q) I believe that the user-defined FPGA registers don't exist in the verilog code. Do I need to add them manually? I was going to do that in usrp_std.v.

A) Correct. The actual registers are not instantiated (they would get pruned out anyway during synthesis if you did.) If you are writing custom FPGA code, you can instantiate any number of:
setting_reg #(FR_USER_XX) user_reg(...);
...in whatever verilog module you create, and just be sure to include:
usrp/firmware/include/fpga_regs_standard.h.
The path in the include statement will vary depending on where your verilog module is in the file system. Also, if you are adding setting registers to existing modules, then the “include” may already be done it for you.
Finally, the constants FR_USER_0 through FR_USER_15 are available from C++ by including:
usrp/firmware/include/fpga_regs_standard.h
(again, path may vary.)

Q) Is there a way to read from the FPGA registers from usrper?

A) No.

>From python:

```
  # returns 32-bit int
  v = u._read_fpga_reg(regno)
```

Note that there are only 4 readable registers, and I strongly suggest that you use
u.read_io(which_dboard) instead of directly reading register 1 or 2.

```
regno   result
-----   -------------------------
  1     (io_rx_a << 16) | io_tx_a      # read daughterboard i/o pins
  2     (io_rx_b << 16) | io_tx_b      # read daughterboard i/o pins
  3     returns const 0xaa55ff77       # don't count on this
  4     returns const 0xf0f0931a       # don't count on this
```

FYI, the "normal" python interface to the usrp is defined in gr-usrp/src/usrp1.i

Digital Down Converter Questions

Q) In the FPGA DDC, are I and Q samples being generated from complex samples? The adc_interface module just seems to multiplex the same complex sample to 2 lines?

A) The ADC interface module takes care of demuxing the received signal into I and Q signals. Then every pair of IQ is routed to a DDC.

Q) What is the purpose of multiplexer in the FPGA receive path?

A) The MUX is like a router or a circuit switcher. It determines which ADC (or constant zero) is connected to each DDC input. There are 4 DDCs. Each has two inputs. We can control the MUX using `usrp.set_mux()` method in Python.
The Mux value is calculated by:

```
   3                   2                   1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-------+-------+-------+-------+-------+-------+-------+-------+
|   Q3  |   I3  |   Q2  |   I2  |   Q1  |   I1  |   Q0  |   I0  |
+-------+-------+-------+-------+-------+-------+-------+-------+
```

Each 4-bit I field is either ADC 0,1,2,3
Each 4-bit Q field is either ADC 0,1,2,3 or 0xf (input is const zero)
All FPGA DDC Q's must be 0xf or none of them may be 0xf.

We tell each input DDC input (I0, Q0, I1 ... I3, Q3) which ADC is connected to it by using 4 bits (0, 1, 2, 3 or 0xf). So a 32-bit integer would be enough for all 8 inputs to know which ADC is connected. Of course an integer in hexadecimal system will be more convenient if we want to use the `set_mux()` method. For most real sampling applications, the Q input of each DDC is constant zero. So quite often we don't need to modify the standard configuration of the FPGA. Actually it is anticipated that the majority of USRP users will never need to use anything other than the standard FPGA configuration.

Q) I want to insert a filter FIR pass band after the output of ADC and before the input of DDC. What's the sampling frequency for this filter FIR? Is correct to insert this filter?

A) The sample rate of the ADC is 64Msps. The samples then go through the CORDIC to generate complex pairs, then through a CIC with minimum decimation of 4, and then through a halfband 2:1 decimating filter. The lack of hardware multipliers inside the FPGA requires you to run a relatively simple filter unless you are running at "low" sample rates which allow you to serialize the data going through one or two multipliers and accumulate the output.

Q) How the FPGA halfband filter implemented, and why we cannot decimate by less than 8 in standard FPGA configuration?

A) It has 31 taps. The current implementation of the halfband filter in the FPGA requires 8 clocks to process the 31 taps. See the comments about timing in fpga/sdr_lib/hb/halfband_decim.v
This works fine with decimation 8 or higher. To run at decimation 4, you'll need to use a build of the FPGA that does not contain the halfband filter, or you'll need to re-implement the halfband, such that it uses two multipliers instead of one. In addition to the standard 2 Rx (with halfband) 2 Tx build, the *current* code contains an FPGA image with 4 RX paths (without the halfband) and with 0 transmit paths. It's not installed by default.

Q) ADC samples at 64MHz, and passes through both I and Q channels over the 24-bit RX bus. Internal to the FPGA, the CIC automatically decimates by a value of at least 4. The halfband decimating FIR filter internal to the FPGA decimates by a fixed value of 2. This gives a minimum decimation rate of 8, leaving 8Msps going over the USB of the USRP. Is this correct?

A) Yes. Note that some FPGA builds don't contain the half-band.
With 16-bit I & Q decim = 8 -> 8MS/sec -> 32MB/sec.
With 8-bit I & Q decim = 4 -> 16MS/sec -> 32MB/sec

Q) Is there any other way to decrease the sample rate beside the USRP decim_rate?

A) The decim_rate sets how much decimation is done in the FPGA. You can of course perform additional decimation in software. You'd probably want to use gr.fir_filter_ccf for that job. The first argument is the decimation rate.

Note:
To use 4 channels, you must use the std_4rx_0tx.rbf FPGA image. If you use the std_4rx_0tx.rbf image, the decimation rate must be <= 128. If you need more decimation, you must do it in software.

Q) Are odd decimation numbers ok? Can I use a decimation of 125?

A) With the std_4rx_0tx.rbf image, I believe odd numbers will work. They will not work with std_2rxhb_2tx.rbf

Q) I wonder how to calculate the frequency resolution of the USRP CORDIC algorithm?

A) This paper is really good for understanding the CORDIC:
http://web.njit.edu/~hkj2/CORDIC.pdf

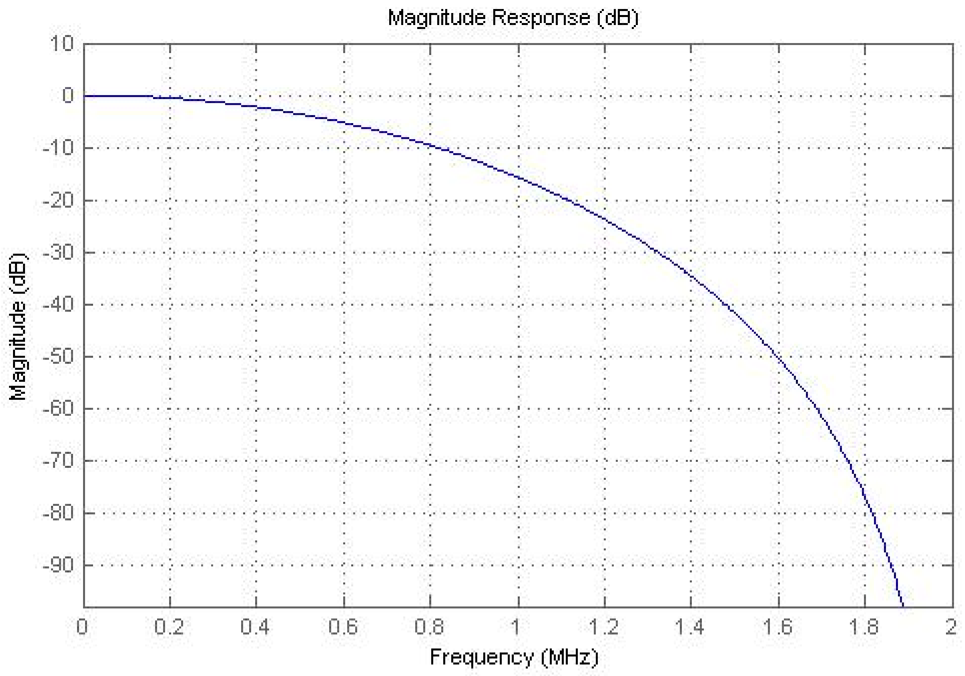
It is specifically written to look at FPGA implementations, which is nice. As I understand it, the USRP uses the CORDIC as described in section 3.1 of that paper. A phase accumulator is used to spin the angle around, and the modulated sin/cos or xi is the output on xo and yo after 12 iterations of the algorithm. The value of zo should be zero, and any error leftover should be represented on that output. The resolution should really be how slowly you can spin the zi component while maintaining accuracy out of the CORDIC. It may be that with 12 iterations and 16-bit inputs 0.01 Hz is possible, whereas more iterations or larger inputs might get better resolution, but I suspect you're really past the point of diminishing returns at that point.

The "phase generator" part of the CORDIC block works by incrementing a 32-bit phase register by a fixed amount per clock cycle. The full size of the register represents 2*PI() of phase, or one cycle of the waveform. The user programmed phase increment per clock cycle then represents frequency.

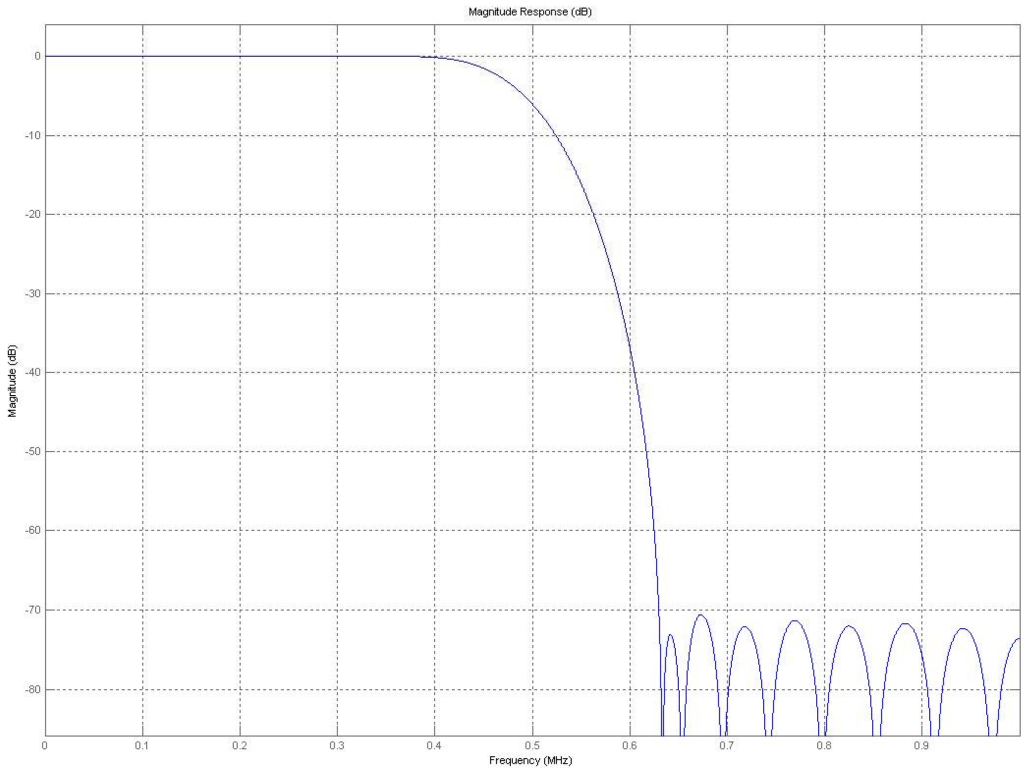
In the receive chain of the FPGA, the phase generator is clocked at 64 MHz. Thus, the minimum delta-frequency (a one bit change in the phase increment register) is 64 MHz / pow(2, 32) = 0.0149 Hz.
Thus, for DC, the phase increment value is zero, for 0.0149 Hz, it is 1, for 0.0298 Hz, it is 2, all the way up to 32 MHz, where it is pow(2, 31). You can also tune negative frequencies, where -1 creates -0.0149Hz, etc.
The CORDIC block then uses the resulting "sawtooth" phase value to rotate the incoming signal by that amount, resulting in complex frequency conversion. The FPGA CORDIC deals with phase and not frequency, so there is no concept of frequency resolution for it. The one being used in the USRP has 14 bits of PHASE resolution. Frequency resolution is controlled by the NCO, or phase accumulator. In our case it has 32 bits of resolution. 64 MHz/2^32 is your frequency resolution in Hz.

Q) If we want a decimation of 64, what is the response of the CIC filter (decimation by 32), HBF (decimation by 2) and the cascaded CIC+HBF (total decimation 64) in the DDC?

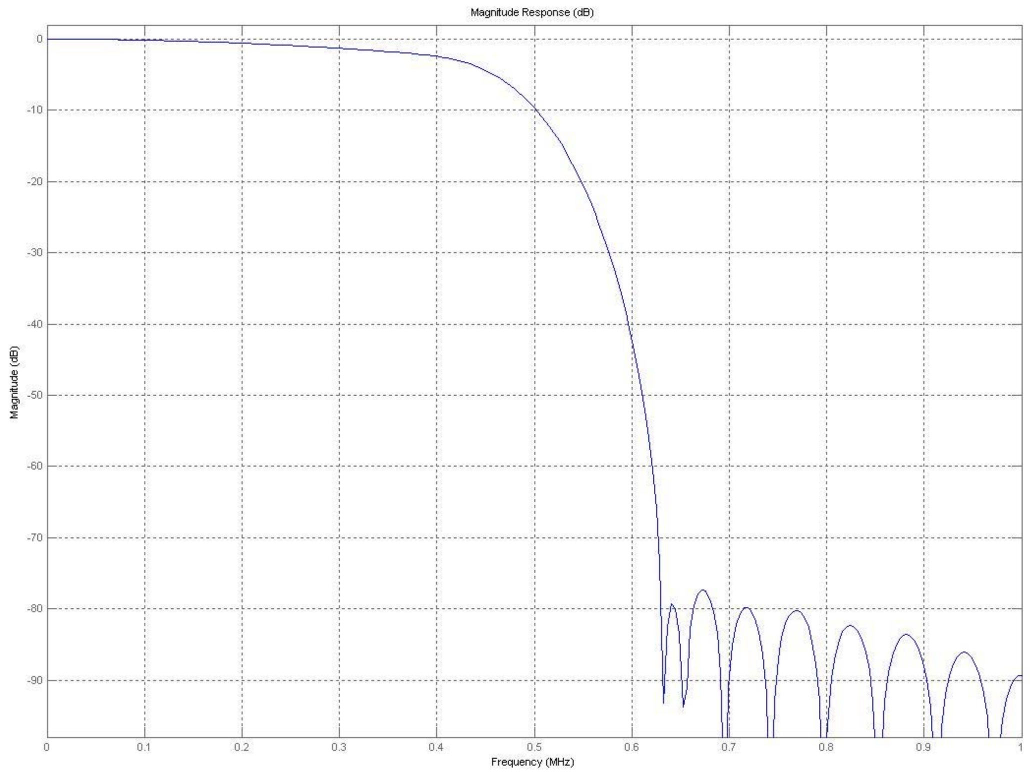
A) For decimation 32, the response of the CIC (4 stages) frequency response from 0 to 2MHz is shown below:



The HBF frequency response is shown below (normalized frequency):



The cascaded response (CIC + HBF) is shown below (normalized frequency):



Digital UP Converter Questions

Q) So from what I understand, in the transmit side the digital up converters are implemented in the AD9862 chip but the interpolation is done in the FPGA. I believe in the receive side the down conversion is done in the FPGA. So why can't we do the same for the transmit side and implement the Digital UP Converters in the FPGA?

A) On the transmit side we use our own CIC to interpolate from whatever ratio comes over the USB to 32 MS/s. This is sent to the 9862 which then interpolates again by a factor of four to bring us to 128 MS/s. The up converter is in the 9862 as you say. In the receive side, the down conversion is done in the FPGA, because the 9862 does not have a down converter in the RX path. The bus going to the TX path on the 9862 is multiplexed (I then Q samples), and we run it at our normal clock rate of 64 MHz. This means we can only get 32 MS/s complex across it. If we did the up conversion in the FPGA, we would thus be limited to about a 12 MHz carrier.

Q) The AD9862 chip contains interpolation filters as well. Could those be used instead of the CIC interpolators on the FPGA? In that way, both the up conversion and the interpolation would be done in the AD9862 chip and more space would be available on the FPGA?

A) We use those interpolation filters already, but they only do a factor of 4, so we still need a CIC in the FPGA.

Q) What happened to the cic_int_shifter.v file in the USRP source? I am looking at re-enabling up conversion in the FPGA but cannot seem to locate this file.

A) It's still in: usrp/fpga/sdr_lib/cic_int_shifter.v

Q) Is the data being clocked out of the USRP to DAC is at 64Msps?

A) More precisely, there are two interleaved channels, each running at 32MS/s. The AD9862 interpolates each stream by 4, giving two streams at 128MS/s.

Q) There are two points that interpolation can happen - inside the AD9862 and internal to the FPGA. Within the FPGA, the CIC filter is the interpolating structure and has a variable rate, whereas the AD9862 has a fixed interpolation rate of 2x if a real-only signal is being used, or 4x is possible if interleaved with I/Q at 64Msps - giving the sample rate of I/Q 32Msps.Is that correct?

A) Yes.

Q) I am unsure what the minimum interpolating rate of the CIC is, or the maximum for that matter?

A) It's probably 1 or 2, though I doubt it's been tested. The (workable) maximum is determined by the width of the intermediate stages of the CIC. I believe we're good to 128 in the FPGA CIC.

Q) Who sets the interpolation rate of the CIC internal to the FPGA to get from the specified number of samples per symbol from a modulator block in GNU Radio to a number that the CIC can interpolate into 64Msps?

A) The code in usrp_standard.cc controls the FPGA interpolation rate. The rates do not have to be powers of two. From the user point of view the net interpolation rate must be in [4, 512] and a multiple of 4. This is all controlled in usrp_standard.cc.

Q) There are 2 complex interpolators implemented in the FPGA (say int0, and int1). There are 4 real (ie, 2 complex) up converters implemented in the DAC chips (say DAC0, DAC1, DAC2, DAC3). The mux routes I and Q of int0 and int1 to the appropriate DACs.
Is it true that DAC0,1 are hardwired to the TXA side of the USRP and dac2,3 to the TXB? Also, is the second complex interpolator used only if nchannels = 2?

A) Yes.

Q) I found out that the max TX freq is limited in usrp_standard.cc to 44MHz (since USE_FPGA_TX_CORDIC is undefined). Now, since a 128MSps should safely ensure 64MHz, what is actually imposing that limitation?

A) Its set by the passband of the interpolator in the AD9862. The digital up converters in the AD9862 don't allow you to set the frequency to values close to the Nyquist frequencies (from 44MHz to 84MHz). Since it samples at 128 MHz, the prohibited ranges are around multiples of 64 MHz.

Q) I'm having trouble understanding exactly how to go about sending two independent, real-valued signals to different SMA connectors on the same transmit daughterboards?

A) As the software is currently built, you cannot send two independent, real-valued signals to different connectors on the same transmit daughterboard. The reason is that we are using the digital up-converters in the AD9862, and have the AD9862's configured in "Dual Channel Complex DAC Data" mode. See pages 20-21 of the AD9862 datasheet. If you only need two independent TX channels, the quick fix is to use two Basic Tx daughterboards. This is supported. Just tell the USRP that you're using two channels of data, and then provide a stream with two interleaved channels of I & Q data. An additional thought is that if you don't need digital up-conversion, I don't think it would take much to get two independent channels out. You should be able to use the existing FPGA and host code, and just manually jam the right magic values into the AD9862 registers 19 and 20. Make sure that you continue to run the 4x interpolator on the AD9862 so all the data rates remain the same. You can write the AD9862 regs from python using:

```
u = usrp.sink_s(...)
u._write_9862(which_codec, regno, value)     # note the leading _ character
                                              # which_codec is 0 or 1 for side A or B
```

Look in usrp_standard.cc at usrp_standard_tx::usrp_standard_tx to see how we normally setup these registers.

Q) At what rate should I send the samples to the tx_chains? Is every 4 clock cycles? They seem to be controlled by tx_strobe, but I failed to understand what this signal actually mean.

A) The minimum interpolation rate is 4 due to the CIC filter limitations, so once every 4 will yield one side of the spectrum. I believe 128 is the other end of the spectrum, but you should set a constant for the rate so we can test with different ones. The tx_strobe happens once every rate clock cycles - with which it will send in a new sample. That strobe sends a new value down the TX stream and reads a new one ready for the next tx_strobe.

Q) What is the point of the bus_reset and the clear_status signal? They are both input to current tx_buffer.

A) I am not sure of this - but I am assuming to stop/reset the transmit path and clear any status messages respectively.

Q) We have multiple USRPs clock-synchronized, but we haven't found a way to set things up so that the different sample streams hit the D/A converters at the same clock.

A) We have gotten multiple USRPs synchronized for receive. That FPGA code and host code is in the subversion. For transmit, the problem could be a bit more difficult, depending on whether you can deal with a constant phase difference between the transmitting USRPs or not (MIMO vs beam forming). If you need the phases to be identical between the USRPs then you will have to change the verilog such that you are no longer using the digital up converters in the AD9862 and instead implement the DUC in the FPGA. This is because there's no way to directly control the phase accumulator in the AD9862. If you can deal with a constant phase offset, then a strategy similar to what Marin used for Rx should work for TX.

Q) My questions are regarding tx_chain.v. I am a bit confused as to how the flow of the code would proceed. There is no "always" statement in the code.

A) This is just wiring a couple things together to get the transmission chain to work at the full clock rate. There are 3 main components here: a phase accumulator, a CORDIC and an interpolating CIC filter. The baseband signals are fed into the CORDIC along with a phase from the phase accumulator. For each sample that goes in, the phase accumulator rotates. This generates an IF that is mixed with your baseband signal. The interpolating CIC filter gets your baseband signal up to the proper sample rate to feed the DACs.

Digital I/O Pins Control Questions

Q) I am using the daughterboard's DEBUG IO pins, and I am connecting them to my logic analyzer. I have to enter the threshold value in my logic analyzer. I can choose between TTL (1.4V) CMOS (2.5V) ECL (- 1.3V). I am not sure which one to choose. Right now I have it set to a user value of 3.30V, is that the right choice?

A) All the signals are LVCMOS, 0 to 3.3V. The correct threshold is 1.65V (3.3V/2)

Q) I generated a signal using the usrp_siggen.py function and tried to use the IO pins on the basic TX board to monitor the digital output on a logic analyzer but it seems that no signal goes to those pins. Is there something I need to change in the verilog code to be able to use the debug IO pins?

A) There are two ways to control the I/O pins, either from the host, or from within the FPGA.

Method A:
To control them from the host no changes are required in the verilog. From python you need to tell it to "output enable" the pins you are interested, and then write whatever values you like to them:

From gr-usrp/src/usrp1.i:

```
  /*!
   * \brief  Write direction register (output enables) for pins that go to daughterboard.
   *
   * \param which_dboard [0,1] which daughterboard
   * \param value value to write into register
   * \param mask which bits of value to write into reg
   *
   * Each daughterboard has 16-bits of general purpose i/o.
   * Setting the bit makes it an output from the FPGA to the daughterboard.
   *
   * This register is initialized based on a value stored in the
   * daughterboard EEPROM.  In general, you shouldn't be using this routine
   * without a very good reason.  Using this method incorrectly will
   * kill your USRP motherboard and/or daughterboard.
   */
  bool _write_oe (int which_dboard, int value, int mask);

  /*!
   * \brief Write daughterboard i/o pin value
   *
   * \param which_dboard [0,1] which daughterboard
   * \param value value to write into register
   * \param mask which bits of value to write into reg
   */
  bool write_io (int which_dboard, int value, int mask);

  /*!
   * \brief Read daughterboard i/o pin value
   *
   * \param which_dboard [0,1] which daughterboard
   * \returns register value if successful, else READ_FAILED
   */
  int read_io (int which_dboard);
```

E.g.,

```
 # Assumes Basic_Tx in slot A.  Do not do this blindly!  Output enabling all the i/o pins
 # on other daughterboards will cause problems (burn up daughterboard  and/or FPGA)

 u = usrp.sink_c(0, 64)
 side = 0  # side A
 u._write_oe(side, 0xffff, 0xffff) # set all i/o pins as outputs
 counter = 0
 while 1:
        u.write_io(side, counter, 0xffff)
        counter = (counter + 1) & 0xffff
```

Method B:

If however, you're trying to control the debug pins from the FPGA, you'll need to output enable them from the host, and enable them as debug outputs. You do the last step by writing to the FR_DEBUG_ENABLE FPGA register:

From usrp/firmware/include/fpga_regs_common.h:

```
// If the corresponding bit is set, internal FPGA debug circuitry
// controls the i/o pins for the associated bank of daughterboard
// i/o pins.  Typically used for debugging FPGA designs.

#define FR_DEBUG_EN            14
#  define bmFR_DEBUG_EN_TX_A        (1 << 0)      // debug controls TX_A i/o
#  define bmFR_DEBUG_EN_RX_A        (1 << 1)      // debug controls RX_A i/o
#  define bmFR_DEBUG_EN_TX_B        (1 << 2)      // debug controls TX_B i/o
#  define bmFR_DEBUG_EN_RX_B        (1 << 3)      // debug controls RX_B i/o
```

These defines are available in python like this:

```
from usrp_fpga_regs import *
 u = usrp.sink_c(0, 64)
 u._write_oe(0, 0xffff, 0xffff)
 u._write_fpga_reg(FR_DEBUG_EN, bmFR_DEBUG_EN_TX_A)
```

Q) I would appreciate any suggestions on how to connect outputs from the FPGA to the debug pins (such as the io_tx_a output) in verilog?

A) Look at usrp/fpga/toplevel/usrp_std/usrp_std.v:

```
  wire [15:0] reg_0,reg_1,reg_2,reg_3;
  master_control master_control
    ( .master_clk(clk64),.usbclk(usbclk),
      .serial_addr(serial_addr),.serial_data(serial_data),.serial_strobe(serial_strobe),
      .tx_bus_reset(tx_bus_reset),.rx_bus_reset(rx_bus_reset),
      .tx_dsp_reset(tx_dsp_reset),.rx_dsp_reset(rx_dsp_reset),
      .enable_tx(enable_tx),.enable_rx(enable_rx),
      .interp_rate(interp_rate),.decim_rate(decim_rate),
      .tx_sample_strobe(tx_sample_strobe),.strobe_interp(strobe_interp),
      .rx_sample_strobe(rx_sample_strobe),.strobe_decim(strobe_decim),
      .tx_empty(tx_empty),
      //.debug_0(rx_a_a),.debug_1(ddc0_in_i),
      .debug_0(tx_debugbus[15:0]),.debug_1(tx_debugbus[31:16]),
      .debug_2(rx_debugbus[15:0]),.debug_3(rx_debugbus[31:16]),
      .reg_0(reg_0),.reg_1(reg_1),.reg_2(reg_2),.reg_3(reg_3) );
```

The arguments .debug_0(...) through .debug_3(...) are the signals that get connected to the debug pins.

```
 debug_0 -> TX_A
 debug_1 -> RX_A
 debug_2 -> TX_B
 debug_3 -> RX_B
```

Q) How do I assign the pins for the FPGA? I mean the external IO pins.

A) The external IO pins should not have to be mapped if you use Matt's Quartus II project. It will already have the pins mapped to where they have to be mapped.

Q) I have the TV_RX board in slot B and Basic RX in slot A of the USRP. I am trying to get the raw A/D data that comes into the FPGA, out on the daughter card connector. I would like to use the 16-bit general purpose debug pins from the FPGA on the connector on Basic RX. However, I’d like to be careful before I write to the direction register using the _write_oe given that an improper setting can cause damage. My question is whether I can have both TV_RX and Basic RX boards plugged in, while I use the _write_oe? If yes, what is the precaution that I have to follow?

A) You are safe with both the basic boards and the TVRX, since they don't drive any of the pins.

Q) I have an application where I will need the USRP AUX digital I/O to control an antenna switching network at approximately 1 kHz frequencies. How far up the software chain does support for the USRP AUX digital I/O go? Is there much support for them at the python level?

A) There are several ways to do this. All of the daughterboard IO pins can be controlled from software, all the way up to the python application level. However, if you want the pins to change quickly, you need to do it in the FPGA. There is already a mechanism for antenna switching. Look in the auto_tr code in db_flexrf.py. Also you can use the gr_gpio.

Q) What is the gr_gpio?

A) **gr-gpio** is an extension to the normal USRP firmware, implemented as an alternative FPGA bit stream, using the existing USRP host code. With the **gr-gpio** component you can transmit and receive a digital stream to and from the USRP which is aligned with the existing analog stream. Digital data is sent to or received from the daughterboard GPIO pins and sacrifice one bit each from the I and the Q analog streams to transport the digital bits. See gnuradio-wiki.

Q) I would like access to I and Q output data (12 bits each) from ADC U601 using the debug headers from two RFX2400 daughterboards. I Know that:

```
* Each daughterboard has 16-bits of general purpose i/o.
* Setting the bit makes it an output from the FPGA to the daughterboard.
* This register is initialized based on a value stored in the
* daughterboard EEPROM. In general, you shouldn't be using this routine
* without a very good reason. Using this method incorrectly will
* kill your USRP motherboard and/or daughterboard.
*/
bool _write_oe (int which_dboard, int value, int mask);

u = usrp.sink_c(0, 64)
  side0 = 0  # side A
  u._write_oe(side0, 0xffff, 0xffff)    # set all i/o pins as outputs
side1=1  # side B
  u._write_oe(side1, 0xffff, 0xffff)    # set all i/o pins as outputs
```

A) The daughterboard and the FPGA are both driving at least pin 2. Though the pins are called "general purpose i/o", that doesn't mean the user should mess with all of them. Some of them are used by the daughterboard code to control or read status from parts on the daughterboards.

Which pins are safe for the user to mess with depends on the design of the given daughterboard. Hence our scary warning on _write_oe.

Step one:
Look at the schematics for the RFX-2400 board and figure out which of the 16 I/O pins are actually used by the TX and RX halves of the transceiver daughterboard, and which ones are available for your use. See which of the I/O pins actually makes it to the header on the board.

Step two:
Look again and check your work.

Step three:
Look at the daughterboard code (db_flexrf.py). See what it's doing to the various output enables and I/O pins. Does this match your understanding of which pins are available for your use?

Step four:
Confirm again which pins are safe for you to be messing with.
(I believe that the high 7-bits are available for your use. It might be 8, but I haven't looked at the schematics in a _long_ time. I'm not kidding when I say _you_ should check the schematics.)

Now figure out new values for the calls to u._write_oe(...), that only change the OE values of the pins that make it to the header and aren't used for some other purpose by the code that's controlling the daughterboard.

Q) Next output enable the debug pins and enable them as debug outputs. u._write_fpga_reg(FR_DEBUG_EN, ??) I am not sure how to safely output enable the debug outputs. Would 0xf do the job?

A) All the output enables are controlled using _write_oe. FR_DEBUG_EN determines whether the debug pins are routed to the i/o pins or whether the normal write_io and auto T/R values make it to the header. See the bottom of master_control.v for the details.
Given what you'll find out after you look at the schematics, no value of FR_DEBUG_EN except for 0 is safe when you're using two RFX-2400 boards. If you can get by with a single RFX-2400, my suggestion is that you put it on the A-side, and then put a Basic Tx and a Basic Rx on the B-side. Then you've got a total of 32 uncommitted I/O pins on the B-side.

The FR_DEBUG_EN register wasn't designed to solve every possible problem. It solved the one we had, which was getting debug info out to a Basic Rx and/or Basic Tx.

Daughterboards Questions

Q) If I want to apply the output (50 ohm) from a function generator directly to the BasicRX inputs, what is the allowed voltage range that can be applied?

1) Without damaging anything?
2) Without exceeding the range of the ADC?
3) Is the input signals need to be DC offset so as never to go below ground?

A) As follows:
1) 3V p-p should be safe, since it won't exceed the 3.3V supply voltage.
2) The full-scale range of the ADC is 2V p-p.
3) No, the transformer handles the biasing. You put in a signal that is +/-1V.

Q) I have a radio with a 10.7MHz real (not complex) output that I would like to connect to the basic Rx daughter board directly.

A) Just connect the real input to the input connector, and use 0xf0f0f0f0 setting for the mux value of the USRP source. This feeds a zero into each of the DDC Q inputs and ADC0 into the DDC I inputs. For your receiver with the 10.7 MHz IF, set the DDC frequency to -10.7e6 and you'll get complex baseband across the USB. Pick the decimation value to set the width of the the channel you're interested in. The ADC sample rate is 64e6 (adc_freq()). Divide that by the decimation factor and that will give you the sample rate across the USB. The USB samples are 16-bit I and 16-bit Q. i.e., 4 bytes/sample. There's a fourth-order CIC filter in the FPGA, so there's some roll-off across the band.

Q) If we apply maximum none distorted input signal to the BasicRX board, what is the USRP DDC output value?

A) Using high accuracy function generator and USRP with Basic-RX at 0 dB gain with decimation of 8):

a) With Sine wave signal at frequency =250 KHz and **0dBm** input power to BasicRX we get:

Theoretical calculations:
When 0dBm signal injected into 50 Ohm load, then we should have signal RMS = 0.225Volt and Signal voltage P-P = 0.636396103Volt.

USRP readings:
The maximum counting obtained from USRP for this signal was **4845**.

b) With Sine wave signal at frequency =250 KHz and **5dBm** input power to BasicRX we get:

Theoretical calculations:
When +5dBm signal injected into 50 Ohm load, then signal RMS = 0.4Volt and Signal P-P = 1.13137085Volt.

USRP readings:
The maximum counting obtained from USRP for this signal was **8337.**

c) With Sine wave signal at frequency =250 KHz and **9dBm** input power to BasicRX we get:

Theoretical calculations :
When +9dBm signal injected into 50 Ohm load, then signal RMS = 0.64Volt and Signal P-P = 1.81019336Volt.

USRP readings:
The maximum counting obtained from USRP for this signal was **13096.**

d) With Sine wave signal at frequency =250 KHz and **10dBm** input power to BasicRX we get:

Theoretical calculations:
When +10dBm signal injected into 50 Ohm load, then signal RMS = 0.71Volt and Signal P-P = 2.008183259Volt.

USRP readings:
The maximum counting obtained from USRP for this signal was **13570** (Saturated).

From the above results we see that the BASIC-RX is saturated when the input signal power was +10dBm. This is logical, since the maximum input signal to the USRP ADC AD9862 is 2Volt P-P (according to its data sheets). The value obtained from USRP FPGA for input signal of 2V P-P was about 13570.

Q) How RF tuning accomplished in daughterboards?

A) We have to understand the comments about tuning being a "two-step process"? First we ask the front-end to tune as close to the desired frequency as it can (For example, on the RFX boards, the PLL step size is 4 MHz.). Then we use the result of that operation and our target_frequency to determine the value for the digital down converter.

The return value from tune software function is an instance of tune_result which can be examined to see how everything was setup:

1- The baseband_freq is the RF frequency that corresponds to DC in the RF front-end IF output (the input to the A/D's and from there to the digital down-converter). Note that this isn't necessarily the location of the signal of interest. Some daughterboards have the signal of interest at a non-zero IF frequency.
2- The dxc_freq is the frequency value used in the digital down or up converter.
3- The residual_freq is a very small number on the order of 1/100 of a Hz. It can be ignored.
4- The Inverted is true if the spectrum is inverted, and we weren't able to fix it for you.

On the receive path, the end result of tune is that the signal at the given target RF frequency ends up at DC in the complex baseband input from the USRP.

Note:
Abstractly, TX and RX daughterboards can be split into two general categories:

1- Those that use both converters as a pair (boards doing quadrature up or down conversion) and
2- Those that can use the converters independently.

Because we're using the digital up converter in the AD9862 in "Dual Channel Complex DAC Data Mode", both D/A outputs are related (I&Q) and as a result, all TX daughterboards are "quadrature" boards.
On the receive side we've got a bit more variation. The TV_RX board (for example) is *not* a quadrature board, since it only uses a single A/D.
On the RFX transceiver boards (400, 900, 1200 and 2400), both the Tx side and Rx side are quadrature.

Q) What is a subdevice?

A) The TX or RX daughterboard consists of either one or two subdevices. Quadrature boards have a single subdevice. Non-quadrature boards have one or two subdevices and it is depending on the daughterboard design.
The fundamental capabilities of a subdevice are the ability to tune and set gain. Each subdevice has a daughterboard specific class that is derived from db_base.py (in gr-usrp).
The visible methods are:

```
def dbid(self):
  """
  Returns integer daughterboard ID from EEPROM
  """

def name(self):
  """
  Name of daughterboard.  E.g., "TV Rx"
  """

def freq_range(self):
  """
  Return range of frequencies in Hz that can be tuned by this daughterboard.

  @returns (min_freq, max_freq, step_size)
  @rtype tuple
  """

def set_freq(self, target_freq):
  """
```

```
        Set the frequency.

        @param freq:  target RF frequency in Hz
        @type freq:   float

        @returns (ok, actual_baseband_freq) where:
           ok is True or False and indicates success or failure,
           actual_baseband_freq is the RF frequency that corresponds to DC in
the IF.
        """

    def gain_range(self):
        """
        Return range of gain that can be set by this daughterboard.

        @returns (min_gain, max_gain, step_size)
        Where gains are expressed in decibels (your mileage may vary)
        """

    def set_gain(self, gain):
        """
        Set the gain.

        @param gain:  gain in decibels
        @returns True/False
        """

    def is_quadrature(self):
        """
        Return True if this daughterboard does quadrature up or down conversion.
        That is, return True if this board requires both I & Q analog channels.

        This bit of info is useful when setting up the USRP Rx mux register.
        """
```

When we create an instance of a usrp source or sink, we now get an additional attribute, "db", that lets us control the daughterboard subdevices.

```
  u = usrp.source_c(0, 64)

  u.db is a tuple of length 2.

  u.db[0] contains a tuple of the subdevices for "side A"
  u.db[1] contains a tuple of the subdevices for "side B"
```

The tuples of subdevices each have either 1 or 2 elements depending on the daughterboard installed on the respective side. Each subdevice is an instance of a class that's derived from db_base and exports the methods listed above. When we're tuning, there are really at least two knobs to twist:

(1) Some kind of PLL on the daughterboard that determines what RF frequency appears in the IF.

(2) The digital down converter that extracts the band of interest from the digitized IF.

Given a target frequency, we control them both like this:

```
def set_freq(self, target_freq):
    """
    Set the center frequency we're interested in.

    @param target_freq: frequency in Hz
    @rtype: bool

    Tuning is a two step process.  First we ask the front-end to
    tune as close to the desired frequency as it can.  Then we use
    the result of that operation and our target_frequency to
    determine the value for the digital down converter.
    """

    # self.subdev is the subdevice we are controlling
    ok, baseband_freq = subdev.set_freq(target_freq)

    ddc_freq, inverted = usrp.calc_dxc_freq(target_freq, baseband_freq, self.u.converter_rate())

    ok &= self.u.set_rx_freq(0, ddc_freq)
```

The Local Oscillator (LO) frequency of the RFX boards is set automatically when we set the frequency of reception or transmission. The LO is a multiple of 2 or 4 MHz depending on which RFX board we have. We can change that to be a multiple of 1MHz, but we will have worse phase noise.

Q) In LFRX board, with the input open, I measure 108 mV on the SMA connector and a DC component from the ADC. What is the problem?

A) If you connect a 50 ohm resistor across the SMA, you do indeed get 62mV at the connector, but 0V at the ADC, which is what we really care about. If you leave the SMA open, you do get a small voltage being read at the ADC.
So yes, the LFRX does have DC bias on its input, but as long as you drive it with a 50 ohm load, you get the right answer at the ADC. The LFRX wasn't designed for measuring DC voltages, although it can as long as you have a 50 ohm source. And if your source is not 50 ohms, you can do the calculations necessary to scale the gain. Also, please note that the amp is inverting, so if you put 1V through 50 ohms into the SMA, the ADC will tell you that it is -1V.

Q) The gEDA PCB layout program returns illegal file format when opening RFX boards, however, it reads the basic RX/TX daughter boards, why?

A) The RFX daughterboard PCBs were designed using a commercial program called PADS, not using PCB from the gEDA suite.

Q) Can any one tell me how far apart the SMA connectors are set on the basic TX daughter board; I'm trying to design a PCB to match up with it.

A) 0.570 inches

Q) How we can know what our daughterboard type is?

A) You can find out what version you have by running print-db in the usrp/host/apps directory.

Q) What is involved in changing an RFX2400 to an RFX1200?

A) To convert to an RFX1200, you need to:

- Cut the traces which FIL1
- Put a capacitor in C204, anything between about 50pF and 1000pF is fine, size 0603
- Put the board on side A of a USRP, power it up, and reburn the EEPROM using the command usrp/host/apps/burn-db-eeprom -A -f -t rfx1200_mimo_b

You can also change some of the final amp tuning components if the power out is a little low.This is not always necessary, but sometimes gets another dB of extra power.

Q) What is involved in changing an RFX900 to an RFX1800?

To convert the you only have to do 3 things:

1. Cut the traces which go to FIL1.
2. Put a capacitor in C204.
3. Run the following command:
```
./burn-db-eeprom -A --force -t rfx1800_mimo_b
```

If you cut the filter out you need to complete the signal path by populating C204 with a capacitor in roughly the 100pF to 10,000pF range (#FIXME#). Very high capacitor values have bad parasitic effects -- 0.1uF is fine but don't use 10uF. To turn it back into a RFX900 again, or to turn your RFX1800 into a RFX900, you can run the command:
```
./burn-db-eeprom -A --force -t rfx900_mimo_b
```

Q) I couldn't find any documentation on what is stored in the USRP daughterboards EEPROMs?

A) The EEPROMs on the daughterboards basically store an identifier for the type of board, and can store some DC offsets and the like. The daughterboard EEPROM format is defined in usrp/firmware/include/usrp_i2c_addr.h

Q) I would like to develop a user daughter card, to interface with the USRP motherboard, Please give me some clues (pointer to information) to know what should be written in the ROM (EEPROM) and consequences in the devices programming?

A) The format of the ROM is documented in usrp/firmware/include/usrp_i2c_addr.h. Just be sure that bytes 0x00 through 0x02 make sense. You can set the remainder to 0x00, but be sure to set the checksum correctly in 0x1f. There's a script that will do this for you over the USB. See usrp/host/apps/burn-db-eeprom.

The relevant portion is:

```
// format of daughterboard EEPROM
// 00: 0xDB code for ``I'm a daughterboard''
// 01:   .. Daughterboard ID (LSB)
// 02:   .. Daughterboard ID (MSB)
// 03:   .. io bits  7-0 direction (bit set if it's an output from motherboard)
// 04:   .. io bits 15-8 direction (bit set if it's an output from motherboard)
// 05:   .. ADC0 DC offset correction (LSB)
// 06:   .. ADC0 DC offset correction (MSB)
// 07:   .. ADC1 DC offset correction (LSB)
// 08:   .. ADC1 DC offset correction (MSB)
// ...
// 1f:   .. negative of the sum of bytes [0x00, 0x1e]

#define DB_EEPROM_MAGIC 0x00
#define  DB_EEPROM_MAGIC_VALUE 0xDB
#define DB_EEPROM_ID_LSB 0x01
#define DB_EEPROM_ID_MSB 0x02
#define DB_EEPROM_OE_LSB 0x03
#define DB_EEPROM_OE_MSB 0x04
#define DB_EEPROM_OFFSET_0_LSB 0x05 // offset correction for ADC or DAC 0
#define DB_EEPROM_OFFSET_0_MSB 0x06
#define DB_EEPROM_OFFSET_1_LSB 0x07 // offset correction for ADC or DAC 1
#define DB_EEPROM_OFFSET_1_MSB 0x08
#define DB_EEPROM_CHKSUM 0x1f
```

```
#define DB_EEPROM_CLEN 0x20 // length of common portion of EEPROM
```

You can use the existing "Experimental Rx" daughterboard id, 0xffff, or define a new one. See usrp/host/lib/legacy/usrp_dbid.dat

The dbid is read from the daughterboards and is use to instantiate the correct daughterboard code. You can see what you've got by calling u.daughterboard_id(0) and u.daughterboard_id(1) to retrieve the dbid's from slots 0 and 1.
To talk to your daughterboard using our standard interface, you'll need to write a bit of python. Take a look at gr-usrp/src/db_*.py. Start with db_base.py, then maybe db_basic.py

C++ Interfacing Questions

Q) How we can write Linux C++ application program to interface with USRP?

A) Here it how we can do it:
1- The gnuradio project provides a complete USRP interface library. When compiling the interface program, these library must be linked with our code by using G++ library command (-lusrp).
2- To show the used procedure, suppose we have written a C++ USRP interface program called usrp_test_c++.cpp. To compile this program, we use the following command:

```
$ g++ usrp_test_c++.cpp –o testusrp –lusrp
```

Where:
g++ is the GNU C++ compiler
usrp_test_c++.cpp is our c++ source file
testusrp is our output file
-lusrp is the USRP C++ library

Note:
The following USRP two header files should be in the same compilation directory:
fpga_regs_common.h
fpga_regs_standard.h

3- To run the output file from Linux command line:
./testusrp

Q) Is there a simple C++ USRP interfacing demonstration program?

A) Below is the list for such program

```
// Simple C++ USRP interfacing demonstration program
//
//
// This program was derived and modified from test_usrp_standard_rx.cc

/* -*- c++ -*- */
/*
 * Copyright 2003,2006,2007,2008 Free Software Foundation, Inc.
 *
 * This file is part of GNU Radio
 *
 * GNU Radio is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 3, or (at your option)
 * any later version.
 *
 * GNU Radio is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with GNU Radio; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110-1301, USA.
 */


#include "usrp_standard.h"

// Dumy Function to process USRP data
void process_data(int *buffer)
{
}

#define SAMPELS_PER_READ       (512)              // Must be a multiple of 128

int
main (int argc, char **argv)
{
  bool    loopback_p = false;
  bool    counting_p = false;
  bool    width_8_p = false;
  int     which_board = 0;
  int     decim = 8;                     // 32 MB/sec
  double center_freq = 0;
  int     fusb_block_size = 0;
  int     fusb_nblocks = 0;
  int     nchannels = 1;
  int     gain = 0;
  int     mode = 0;
  int     noverruns = 0;
  bool    overrun;
  int     total_reads = 10000;
  int     i;
  int     buf[SAMPELS_PER_READ];
  int     bufsize = SAMPELS_PER_READ*4;


 if (loopback_p)    mode |= usrp_standard_rx::FPGA_MODE_LOOPBACK;
if (counting_p)    mode |= usrp_standard_rx::FPGA_MODE_COUNTING;



 usrp_standard_rx *urx =  usrp_standard_rx::make (which_board, decim, 1, -1, mode,
                                  fusb_block_size, fusb_nblocks);

 if (urx == 0)
          {
          fprintf (stderr, "Error: usrp_standard_rx::make\n");
          exit (1);
          }

 if (width_8_p)
{
   int width = 8;
   int shift = 8;
   bool want_q = true;
   if (!urx->set_format(usrp_standard_rx::make_format(width, shift, want_q)))
     {
          fprintf (stderr, "Error: urx->set_format\n");
          exit (1);
     }
 }

 // Set DDC center frequency
 urx->set_rx_freq (0, center_freq);

// Set Number of channels
 urx->set_nchannels(1);

// Set ADC PGA gain
 urx->set_pga(0,gain);

// Set FPGA Mux
 urx->set_mux(0x32103210); // Board A only

// Set DDC decimation rate
 urx->set_decim_rate(decim);

// Set DDC phase
 urx->set_ddc_phase(0,0);




 urx->start();              // Start data transfer
```

```
  printf("USRP Transfer Started\n");



        // Do USRP Samples Reading
        for (i = 0; i < total_reads; i++)
        {
                urx->read(&buf, bufsize, &overrun);

                if (overrun)
                        {
                        printf ("USRP Rx Overrun\n");
                        noverruns++;
                        }

                // Do whatever you want with the data
                process_data(&buf[0]);

        }


  urx->stop();  // Stop data transfer
  printf("USRP Transfer Stoped\n");
  delete urx;
  return 0;
}
```

Q) Currently, all USRP daughterboards control is handled from Python. Where I can find the C++ drivers for these boards?

A) The gnuradio team is developing an all C++ daughterboards drivers [I don't know if they finished it]. There are some gnuradio users that developed their own C++ drivers but they did not publish it yet [I Think They Should].
The only published C++ driver was for DBSRX daughterboard by Gregory W Heckler.

Q) Can we use Gregory DBSRX driver directly? How?

A) Yes. See the program below (From Gregory web site www.gps-sdr.com). This is the Main.cpp program. The program, implements a thread safe routines for data recording from the USRP to harddisk drive. The program is listed here for a documentation purpose.

```
#include <unistd.h>
#include <errno.h>
#include <ctype.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <fcntl.h>
#include <iostream>
#include <stdio.h>
#include <math.h>
#include <pthread.h>
#include <sched.h>
#include "usrp_prims.h"
#include "usrp_spi_defs.h"
#include "usb.h"
#include "usrp_standard.h"
#include "usrp_bytesex.h"
#include "fpga_regs_common.h"
#include "fpga_regs_standard.h"
#include "usrp_i2c_addr.h"
#include "db_dbs_rx.h"

using namespace std;

typedef struct _CPX
{
        short r;
        short i;
} CPX;

typedef struct _options
{
        int     realtime;
        int             board_a;
        int             board_b;
        int     decimate;
        int     soft_decimate;
        int     integrate;
        int             verbose;
        int     shift;
        double  ga;
        double gb;
        double  gi;
        double  gr;
        double  f_lo;
        double  f_ddc;
        double  seconds;
        double  bandwidth;
        char    filename[1024];

} options;


//ATCHUNG!!!!!!!!!!!!!!!!!!!
//Set this to the mainboard's clock frequency
//#define F_SAMPLE              (65.536e6)
#define F_SAMPLE                (64.0e6)

#define F_SAMPLE_NOM            (64e6)
#define DDC_FIX                 (F_SAMPLE_NOM/F_SAMPLE)
#define SAMPS_PER_READ          (2048)
#define READ                            (0)
#define WRITE                           (1)

void PostStatus(char *_str) {fprintf(stdout,"%s",_str); fflush(stdout);}
int record(options *_opt);
void *disk_thread(void *_opt);
void *integrate_thread(void *_arg);
void *key_thread(void *_arg);

int disk_pipe[2];
int integrate_pipe[2];
bool grun;

void usage(char *_str)
{

        fprintf(stderr, "usage: [-R] [-s] [-A] [-B] [-o] [-d] [-D] [-f] [-gr] [-gi] [-l] [-w]\n");
        fprintf(stderr, "[-R] run in realtime (data written to named pipe)\n");
```

```
		fprintf(stderr, "[-s] <seconds> to record\n");
		fprintf(stderr, "[-i] <samples> integration period\n");
		fprintf(stderr, "[-A] record from RX board A\n");
		fprintf(stderr, "[-B] record from RX board B\n");
		fprintf(stderr, "[-o] <filename> output file name\n");
		fprintf(stderr, "[-D] <decimation> software decimation (1-16) \n");
		fprintf(stderr, "[-d] <decimation> USRP decimation (8-128)\n");
		fprintf(stderr, "[-f] <frequency> digital downconvert frequency\n");
		fprintf(stderr, "[-ga] <gain> set IF gain for board A (Basic RX only)\n");
		fprintf(stderr, "[-gb] <gain> set IF gain for board B (Basic RX only)\n");
		fprintf(stderr, "[-gr] <gain> set rf gain in dB (DBSRX only)\n");
		fprintf(stderr, "[-gi] <gain> set if gain in dB (DBSRX only)\n");
		fprintf(stderr, "[-l] <frequency> RF downconvert frequency (DBSRX only)\n");
		fprintf(stderr, "[-w] <bandwidth> bandwidth of lowpass filter (BDSRX only)\n");
		fflush(stderr);

		exit(1);
}

void echo_options(options *_opt)
{

		/* Step one, find out what daughterboards are populated */
		/* Open the base RX class */
		usrp_standard_rx *urx = usrp_standard_rx::make(0, 8, 1, -1, 0, 0, 0);
		if(urx == NULL)
		{
			printf("usrp_standard_rx::make FAILED\n");
			return;
		}

		switch(urx->daughterboard_id(0))
		{
			case 1:
				fprintf(stdout,"Board A:\t\tBasic RX\n"); break;
			case 2:
				fprintf(stdout,"Board A:\t\tDBS RX\n"); break;
			default:
				fprintf(stdout,"Board A:\t\tUnknown\n"); break;
		}

		switch(urx->daughterboard_id(1))
		{
			case 1:
				fprintf(stdout,"Board A:\t\tBasic RX\n"); break;
			case 2:
				fprintf(stdout,"Board A:\t\tDBS RX\n"); break;
			default:
				fprintf(stdout,"Board A:\t\tUnknown\n"); break;
		}

		fprintf(stdout, "Realtime\t\t%d\n",_opt->realtime);
		fprintf(stdout, "Seconds:\t\t%f\n",_opt->seconds);
		fprintf(stdout, "Integration:\t\t%d\n",_opt->integrate);
		fprintf(stdout, "Board A Record:\t\t%d\n",_opt->board_a);
		fprintf(stdout, "Board B Record:\t\t%d\n",_opt->board_b);
		fprintf(stdout, "Filename:\t\t%s\n",_opt->filename);
		fprintf(stdout, "Soft Decimation:\t%d\n",_opt->soft_decimate);
		fprintf(stdout, "USRP Decimation:\t%d\n",_opt->decimate);
		fprintf(stdout, "DDC Frequency:\t\t%f\n",_opt->f_ddc);
		fprintf(stdout, "Basic RX Gain:\t\t%f\n",_opt->ga);
		fprintf(stdout, "Basic RX Gain:\t\t%f\n",_opt->gb);
		fprintf(stdout, "RF Gain:\t\t%f\n",_opt->gr);
		fprintf(stdout, "IF Gain:\t\t%f\n",_opt->gi);
		fprintf(stdout, "DBSRX LO:\t\t%f\n",_opt->f_lo);
		fprintf(stdout, "DBSRX Bandwidth:\t%f\n",_opt->bandwidth);

		delete urx;

}

int main(int argc, char **argv)
{
```

```
options record_options;
char *parse;

/* Set default recording options */
record_options.realtime = 0;
record_options.board_a = 1;
record_options.board_b = 0;
record_options.decimate = 32;
record_options.soft_decimate = 1;
record_options.integrate = 0;
record_options.verbose = 0;
record_options.shift = 0;
record_options.gr = 40;
record_options.gi = 10;
record_options.ga = 0;
record_options.gb = 0;
record_options.f_lo = 1.57542e9;
record_options.f_ddc = 0;
record_options.seconds = 1.0;
record_options.bandwidth = 4.0e6;
strcpy(record_options.filename, "data");

for(int lcv = 1; lcv < argc; lcv++)
{
        switch (argv[lcv][1])
        {
                case 's':
                        lcv++;
                        if(isdigit(argv[lcv][0]))
                                record_options.seconds = strtod(argv[lcv], &parse);
                        else
                                usage (argv[0]);
                        break;

                case 'i':
                        lcv++;
                        if(isdigit(argv[lcv][0]))
                                record_options.integrate = strtol(argv[lcv], &parse, 10);
                        else
                                usage (argv[0]);

                        break;

                case 'A':
                        record_options.board_a = 1;
                        break;

                case 'B':
                        record_options.board_b = 1;
                        break;

                case 'o':
                        lcv++;
                        strcpy(record_options.filename, argv[lcv]);
                        break;

                case 'd':
                        lcv++;
                        if(isdigit(argv[lcv][0]))
                                record_options.decimate = strtol(argv[lcv], &parse, 10);
                        else
                                usage (argv[0]);

                        break;

                case 'D':
                        lcv++;
                        if(isdigit(argv[lcv][0]))
                                record_options.soft_decimate = strtol(argv[lcv], &parse, 10);
                        else
                                usage (argv[0]);

                        break;
```

```
		case 'f':
			lcv++;
			record_options.f_ddc = strtod(argv[lcv], &parse);
			break;

		case 'g':
			if(argv[lcv][2] == 'r')
			{
				lcv++;
				if(isdigit(argv[lcv][0]))
					record_options.gr = strtod(argv[lcv], &parse);
				else
					usage (argv[0]);
				break;
			}
			else if(argv[lcv][2] == 'i')
			{
				lcv++;
				if(isdigit(argv[lcv][0]))
					record_options.gi = strtod(argv[lcv], &parse);
				else
					usage (argv[0]);
				break;
			}
			else if(argv[lcv][2] == 'a')
			{
				lcv++;
				if(isdigit(argv[lcv][0]))
					record_options.ga = strtod(argv[lcv], &parse);
				else
					usage (argv[0]);
				break;
			}
			else if(argv[lcv][2] == 'b')
			{
				lcv++;
				if(isdigit(argv[lcv][0]))
					record_options.gb = strtod(argv[lcv], &parse);
				else
					usage (argv[0]);
				break;
			}
			else
				usage(argv[0]);
			break;

		case 'l':
			lcv++;
			if(isdigit(argv[lcv][0]))
				record_options.f_lo = strtod(argv[lcv], &parse);
			else
				usage (argv[0]);
			break;

		case 'w':
			lcv++;
			if(isdigit(argv[lcv][0]))
				record_options.bandwidth = strtod(argv[lcv], &parse);
			else
				usage (argv[0]);
			break;

		case 'v':
			record_options.verbose = 1;
			break;

		case 'R':
			record_options.realtime = 1;
			break;

		default:
			usage (argv[0]);
	}
```

```
        }

        struct sched_param param;
        sched_getparam(0, &param);
        param.__sched_priority = 99;
        sched_setparam(0, &param);

        /* Echo the record options */
        echo_options(&record_options);

        /* Call the recording function */
        record(&record_options);

        exit(1);
}


int record(options *_opt)
{
        pthread_t pdisk_thread;
        pthread_t pintegrate_thread;
        pthread_t pkey_thread;
        db_dbs_rx *dbs_rx_a = NULL;
        db_dbs_rx *dbs_rx_b = NULL;
        double ddc_correct = 0;
        int buff[SAMPS_PER_READ];
        int buffsize = SAMPS_PER_READ*4;
        int lcv, total_reads, k, lcv2;
        int npipe;
        bool overrun;

        /* Make the URX */
        usrp_standard_rx *urx = usrp_standard_rx::make(0, _opt->decimate, 1, -1, 0, 0, 0);
        if(urx == NULL)
        {
                printf("usrp_standard_rx::make FAILED\n");
                return(-1);
        }

        /* N channels according to which boards are enabled */
        urx->set_nchannels(_opt->board_a + _opt->board_b);

        /* Create DB_DBS_RX object if the daughter board ID matches */
        if(urx->daughterboard_id(0) == 2)
        {
                dbs_rx_a = new db_dbs_rx(urx, 0);
                dbs_rx_a->bandwidth(_opt->bandwidth);
                dbs_rx_a->if_gain(_opt->gi);
                dbs_rx_a->rf_gain(_opt->gr);
                dbs_rx_a->tune(_opt->f_lo);

                if(_opt->verbose)
                {
                        printf("DBS-RX A BW: \t\t%f\n",dbs_rx_a->bw());
                        printf("DBS-RX A LO: \t\t%f\n",dbs_rx_a->freq());
                        printf("DBS-RX A IF Gain: \t%f\n",dbs_rx_a->if_gain());
                        printf("DBS-RX A RF Gain: \t%f\n",dbs_rx_a->rf_gain());
                        printf("DBS-RX A Diff: \t\t%f\n",dbs_rx_a->freq()-_opt->f_lo);
                }

                /* Add additional frequency to ddc to account for imprecise LO programming */
                ddc_correct = dbs_rx_a->freq() - _opt->f_lo;
        }
        else
        {
                urx->set_pga(0, _opt->ga);
        }

        /* Create DB_DBS_RX object if the daughter board ID matches */
        if(urx->daughterboard_id(1) == 2)
        {
                dbs_rx_b = new db_dbs_rx(urx, 1);
                dbs_rx_b->bandwidth(_opt->bandwidth);
                dbs_rx_b->if_gain(_opt->gi);
```

```
		dbs_rx_b->rf_gain(_opt->gr);
		dbs_rx_b->tune(_opt->f_lo);

		if(_opt->verbose)
		{
			printf("DBS-RX B BW: \t\t%f\n",dbs_rx_b->bw());
			printf("DBS-RX B LO: \t\t%f\n",dbs_rx_b->freq());
			printf("DBS-RX B IF Gain: \t%f\n",dbs_rx_b->if_gain());
			printf("DBS-RX B RF Gain: \t%f\n",dbs_rx_b->rf_gain());
			printf("DBS-RX B Diff: \t\t%f\n",dbs_rx_b->freq()-_opt->f_lo);
		}

		/* Add additional frequency to ddc to account for imprecise LO programming */
		ddc_correct = dbs_rx_b->freq() - _opt->f_lo;
	}
	else
	{
		urx->set_pga(2, _opt->gb);
	}


	/* Set the mux */
	if((_opt->board_a + _opt->board_b) > 1)
		urx->set_mux(0x32103210);	//for both channels
	else if(_opt->board_a)
		urx->set_mux(0x32103210);	//board A only
	else
		urx->set_mux(0x10321032);	//board B only

	/* Set the decimation */
	urx->set_decim_rate(_opt->decimate);

	/* Set the DDC frequency */
	_opt->f_ddc += ddc_correct;
	_opt->f_ddc *= DDC_FIX;

	if(_opt->f_ddc > (F_SAMPLE_NOM/2.0))
		_opt->f_ddc = F_SAMPLE_NOM - _opt->f_ddc;

	urx->set_rx_freq(0, _opt->f_ddc);
	urx->set_rx_freq(1, _opt->f_ddc);

	/* Reset DDC phase to zero */
	urx->set_ddc_phase(0, 0);
	urx->set_ddc_phase(1, 0);

	printf("DDC 0: \t\t\t%f\n",urx->rx_freq(0));
	printf("DDC 1: \t\t\t%f\n",urx->rx_freq(1));

	/* The number of reads we want */
	total_reads = (int)ceil((float)_opt->seconds*65.536e6/(float)_opt->decimate);
	total_reads /= SAMPS_PER_READ;

	if((_opt->board_a + _opt->board_b) > 1)
		total_reads *= 2;

	grun = true;

	/* Everything set, now create a disk thread & pipe, and do some recording! */
	if(_opt->realtime)
	{
		pthread_create(&pkey_thread, NULL, key_thread, (void *)_opt);

		lcv = mkfifo("/tmp/GPSPIPE", 0666);
		if ((lcv == -1) && (errno != EEXIST))
	    printf("Error creating the named pipe");
	  else
		printf("Named pipe created\n");

		printf("Waiting for client\n");
		npipe = open("/tmp/GPSPIPE", O_WRONLY);
		printf("Client connected\n");

		/* Make pipe write non-blocking, this is to prevent the USRP from overflowing,
		 * which hoses the data steam. It is up to the CLIENT to make sure it is
		 * receiving continguous data packets */
		fcntl(npipe, F_SETFL, O_NONBLOCK);
	}
	else
	{
		/* Everything set, now create a disk thread & pipe, and do some recording! */
		if(!_opt->integrate)
		{
			pipe(disk_pipe);
			pthread_create(&pdisk_thread, NULL, disk_thread, (void *)_opt);
		}

		/* If integrate option is set, start that thread also */
		if(_opt->integrate)
		{
			pipe(integrate_pipe);
			pthread_create(&pintegrate_thread, NULL, integrate_thread, (void *)_opt);
		}
	}


	urx->start();

	printf("USRP Start\n");

	/* Clear out any junk data */
	for(lcv = 0; lcv < 100; lcv++)
		urx->read(&buff, buffsize, &overrun);

	overrun = 0;

	if(_opt->realtime)
	{

		k = 0;

		while(grun)
		{
			urx->read(&buff, buffsize, &overrun);

			write(npipe, &buff, buffsize);

			if(overrun)
			{
				fprintf(stderr, "o");
				fflush(stderr);
			}
		}
	}
	else
	{
		/* Now do actual recording */
		for(lcv = 0; lcv < total_reads; lcv++)
		{
			urx->read(&buff, buffsize, &overrun);

			if(!_opt->integrate)
				write(disk_pipe[WRITE], &buff, buffsize);

			if(_opt->integrate)
				write(integrate_pipe[WRITE], &buff, buffsize);

			if(overrun)
			{
				fprintf(stderr, "o");
				fflush(stderr);
			}
		}
	}

	urx->stop();

	printf("USRP Stop\n");
```

```
        grun = false;

        /* Kill the disk thread */
        buff[0] =  0xdacabafa;

        if(!_opt->realtime)
        {
                if(!_opt->integrate)
                        write(disk_pipe[WRITE], &buff, buffsize);

                if(_opt->integrate)
                        write(integrate_pipe[WRITE], &buff, buffsize);

                /* Wait for thread to return */
                if(!_opt->integrate)
                        pthread_join(pdisk_thread, NULL);

                if(_opt->integrate)
                        pthread_join(pintegrate_thread, NULL);
        }
        else
        {
                pthread_join(pkey_thread, NULL);

                printf("Closing pipe\n");
                close(npipe);
                printf("Pipe closed\n");
        }

        if(dbs_rx_a != NULL)
                delete dbs_rx_a;

        if(dbs_rx_b != NULL)
                delete dbs_rx_b;

        if(urx != NULL)
                delete urx;

        return(1);

}

/* Monitor keyboard to kill real-time thread */
void *key_thread(void *_arg)
{

        int key;

        printf("Key thread start\n");

        while(grun)
        {
                key = getchar();
                printf("%c",(char)key);

                if((char)key == 'Q')
                        grun = false;

        }

        printf("Key thread stop\n");

        pthread_exit(0);

}

void *disk_thread(void *_arg)
{

        options *_opt = (options *)_arg;
        int buff[SAMPS_PER_READ], buff1[SAMPS_PER_READ], buff2[SAMPS_PER_READ];
        char fname1[1024], fname2[1024];
        FILE *fp1, *fp2;
```

```
int nchannel, lcv, k, samps_per_write;
bool _run = true;

printf("Record thread start\n");

/* Set file names */
if(_opt->board_a)
{
        strcpy(fname1, _opt->filename);
        strcat(fname1, ".dba");
        fp1 = fopen(fname1, "wb");
}

if(_opt->board_b)
{
        strcpy(fname2, _opt->filename);
        strcat(fname2, ".dbb");
        fp2 = fopen(fname2, "wb");
}

/* Number of channels */
nchannel = _opt->board_a + _opt->board_b;

/* Number of samples dumped per write */
samps_per_write = SAMPS_PER_READ/_opt->soft_decimate;
if((_opt->board_a + _opt->board_b) > 1)
        samps_per_write /= 2;

while(grun)
{

        // Get data from pipe
        read(disk_pipe[READ], &buff, SAMPS_PER_READ*sizeof(int));

        // Check for thread exit
        if(*((unsigned int*)&buff[0]) == 0xdacabafa)
        {
                _run = false;
                break;
        }

        // Decimate and uninterleave if both boards are being used
        if(nchannel <= 1)
        {
                k = 0;
                for(lcv = 0; lcv < SAMPS_PER_READ; lcv++)
                {
                        buff1[k] = buff[lcv];
                        k++;
                }

                if(_opt->board_a) fwrite(buff1, samps_per_write, sizeof(int), fp1);
                if(_opt->board_b) fwrite(buff1, samps_per_write, sizeof(int), fp2);
        }
        else
        {

                k = 0;
                for(lcv = 0; lcv < SAMPS_PER_READ; lcv+=2)
                {
                        buff1[k] = buff[lcv];
                        buff2[k] = buff[lcv+1];
                        k++;
                }

                if(_opt->board_a) fwrite(buff1, samps_per_write, sizeof(int), fp1);
                if(_opt->board_b) fwrite(buff2, samps_per_write, sizeof(int), fp2);

        }

}

if(fp1) fclose(fp1);
if(fp2) fclose(fp2);
```

```
        printf("\nRecord thread stop\n");

        pthread_exit(0);

}

/*! Integrates and then dumps the resulting I & Q data in int32 format */
void *integrate_thread(void *_arg)
{

        options *_opt = (options *)_arg;
        char fname1[1024], fname2[1024];
        FILE *fp1, *fp2, *fp3;
        int nchannel, lcv, samps_per_read, bytes_per_read;
        int I1, I2, I3, Q1, Q2, Q3;
        int nbytes, bread, nsamps;
        CPX *buff;

        printf("Integrate thread start\n");

        samps_per_read = _opt->integrate;
        if((_opt->board_a + _opt->board_b) > 1)
                samps_per_read *= 2;

        bytes_per_read = samps_per_read*sizeof(CPX);

        buff = (CPX *)malloc(bytes_per_read);

        /* Set file names */
        if(_opt->board_a)
        {
                strcpy(fname1, _opt->filename);
                strcat(fname1, ".pha");
                fp1 = fopen(fname1, "wb");
        }

        if(_opt->board_b)
        {
                strcpy(fname2, _opt->filename);
                strcat(fname2, ".phb");
                fp2 = fopen(fname2, "wb");
        }

        if(_opt->board_a && _opt->board_b)
        {
                strcpy(fname2, _opt->filename);
                strcat(fname2, ".phc");
                fp3 = fopen(fname2, "wb");
        }

        /* Number of channels */
        nchannel = _opt->board_a + _opt->board_b;

        while(grun)
        {

                // Get data from pipe
                nsamps = nbytes = 0;
                while((nbytes < bytes_per_read) && grun)
                {
                        bread = read(integrate_pipe[READ], &buff[nsamps], bytes_per_read-nbytes);

                        if(bread >= 0)
                        {
                                nbytes += bread;
                        }

                        nsamps = nbytes/sizeof(CPX);
                }

                // Check for thread exit
                if(*((unsigned int *)buff) == 0xdacabafa)
                        break;
```

```
		// Decimate and uninterleave if both boards are being used
		if(nchannel <= 1)
		{
			I1 = I2 = Q1 = Q2 = 0;
			for(lcv = 0; lcv < samps_per_read; lcv++)
			{
				I1 += buff[lcv].r;
				Q1 += buff[lcv].i;
			}

			if(_opt->board_a)
			{
				fwrite(&I1, 1, sizeof(int), fp1);
				fwrite(&Q1, 1, sizeof(int), fp1);
			}

			if(_opt->board_b)
			{
				fwrite(&I1, 1, sizeof(int), fp2);
				fwrite(&Q1, 1, sizeof(int), fp2);
			}


		}
		else
		{

			I1 = I2 = I3 = Q1 = Q2 = Q3 = 0;
			for(lcv = 0; lcv < samps_per_read; lcv+=2)
			{
				I1 += buff[lcv].r;
				Q1 += buff[lcv].i;
				I2 += buff[lcv+1].r;
				Q2 += buff[lcv+1].i;
				Q3 += buff[lcv].r * buff[lcv+1].r + buff[lcv].i * buff[lcv+1].i;
				I3 += buff[lcv].r * buff[lcv+1].i - buff[lcv].i * buff[lcv+1].r;

			}

			//if(_opt->board_a && !_opt->board_b)
			if(_opt->board_a)
			{
				fwrite(&I1, 1, sizeof(int), fp1);
				fwrite(&Q1, 1, sizeof(int), fp1);
			}

			//if(_opt->board_b && !_opt->board_a)
			if(_opt->board_b)
			{
				fwrite(&I2, 1, sizeof(int), fp2);
				fwrite(&Q2, 1, sizeof(int), fp2);
			}

			if(_opt->board_a && _opt->board_b)
			{
				fwrite(&I3, 1, sizeof(int), fp3);
				fwrite(&Q3, 1, sizeof(int), fp3);
			}

		}

	}

	if(fp1) fclose(fp1);
	if(fp2) fclose(fp2);
	if(fp3) fclose(fp3);

	free(buff);

	printf("\nIntegrate thread stop\n");

	pthread_exit(0);
```

```
}
```

Q) How we can compile Gregory program (Main.cpp)?

A) We can use:

```
$ g++ db_dbs_rx.cpp Main.cpp -o testdriver -lm  -lusrp –lpthread
```

Note:
The following files should be in the same compilation directory:
db-dbs_rx.cpp
db_dbs_rx.h
fpga_regs_common.h
fpga_regs_standard.h

RSSI Measurement Questions

Q) Is RSSI measuring possible with the USRP motherboard and the RFX transceiver board? I know we have the RSSI circuit onboard, but I don't know on which frequencies it scans?

A) The RSSI on the motherboard will tell you the power within approx. +/- 15 MHz from your carrier. You can also get a digital RSSI which will tell you the power within your signal of interest. The way we have envisioned RSSI is as a 3-part measurement:

1) Analog RSSI (we can read it using AUX ADC)
2) Digital RSSI in FPGA (from output of ADCs)
3) Digital RSSI in host (computed however you like, from the channel zed signal sent over the bus by the USRP)

If you want to use the measured analog RSSI, it will measure whatever passes through the analog channel filter. You can only change that bandwidth by changing inductors and capacitors. Note also that analog RSSI is only on the RFX900, RFX1200, RFX1800, and RFX2400. The RFX400, TVRX, and DBSRX do not have that capability.

Q) Is there an RSSI circuit on these boards? I'm using the Flex 2400.

A) The RFX900, 1200, 1800, and 2400 have an RSSI circuit on board. The RFX400 does not. To read the RSSI value, use read_aux_adc(side,0). The RSSI measures the analog signal level after the lowpass filters on the board. These filters are about 15-20 MHz wide. Thus, anything falling in that band will cause a rise in the RSSI value. It is connected to the low-speed adc AUX_ADC_A1, so you need to read that with read_aux_adc(which_dboard,0) See usrp1.i (and in usrp_basic.{cc, h}):

```
int read_aux_adc (int which_dboard, int which_adc);
```

The RSSI reads the power in the analog baseband signals (power (I) + power (Q)). The analog baseband will contain signals from approximately +/- 15 MHz from the LO frequency.

Q) I can read the analog RSSI indicator with the auxiliary ADCs, but it is not connected to the gain control. Instead, an auxiliary DAC controls the gain control, right?

A) Yes, that is correct. You can reconfigure the board by moving some resistors around to make it automatic if you wanted.

Q) How do I set the mux register to look at the AUX_ADC inputs?

A) You don't use the mux register to read the AUX_ADCs. See usrp/host/lib/usrp_{basic,standard}.h for docs on all of this stuff. See especially usrp_basic_rx::read_aux_adc and usrp_basic_tx::read_aux_adc

```
/*!
 * \brief Read auxiliary analog to digital converter.
 *
 * \param which_dboard     [0,1] which daughterboard
 * \param which_adc        [0,1]
 * \param value            return 12-bit value [0,4095]
 * \returns true iff successful
 */
bool read_aux_adc (int which_dboard, int which_adc, int *value);

/*!
 * \brief Read auxiliary analog to digital converter.
 *
 * \param which_dboard     [0,1] which daughterboard
 * \param which_adc        [0,1]
 * \returns value in the range [0,4095] if successful, else READ_FAILED.
 */
int read_aux_adc (int which_dboard, int which_adc);
```

The second version has a python binding:

```
v = u.read_aux_adc(0, 0)
```

where u is an instance of a usrp.source_c or usrp.sink_c
The bad news is that these are read asynchronously using a slow path over the USB. You probably can't read them fast enough to sample to your input signal, and the sampling is asynchronous.

Motherboard Re-Clocking Questions

Q) I would like to run my USRP off a 44 MHz clock instead of the 64 MHz clock that is currently used. Can I generate a 44 MHz clock from the 64 MHz or do I have to change the crystal? If I change over to the 44 MHz will everything else still work? Especially the DBSRX and RFX2400 boards?

A) You need a new crystal or an external oscillator. You'll need to change the definition of

```
 /*!
  * \brief return frequency of master oscillator on USRP
  */
 long  fpga_master_clock_freq () const { return 64000000; }
```

in usrp_basic.h

Procedure:
I purchased a 44 MHz oscillator from digi-key (p/n 300-7254-1-ND, CSX750FJC44) for $3.38. The first attempt to remove the USRP 64 MHz failed miserably with a pen iron. I purchased a heat gun from digi-key (MA1008-ND) for $52, set it to the 1000 degree setting and the old oscillator came up in about 10 seconds. Installed the new oscillator and checked the 44 MHz clock with a scope. I am using the std_4rx_0tx.rbf FPGA image.
I then tried to run my customized dbs application that records 8-bit I/Q data to a file for me from any of the 802.11b channels. I got all zeros in the data. Then remembered Eric told me to change the clock in db_dbs_rx.py.

My code is a little older than current but here are my modifications:
I gr-usrp/src/db_dbs_rx.py:
1. changed line 88 from
self.refclk_divisor = 16
to
self.refclk_divisor = 11

2. I changed line 108 from
return 64e6/self.refclk_divisor
to
return 44e6/self.refclk_divisor

then in /gr-usrp I did
```
make clean
make
make check
make install
```

Rerun my custom dbs program and it worked fine. I used a decimation of 4 so I'm recording at 11 MSsamples per second.(#FIXME# is this correct ?)

Q) Is there another way to achieve 10 MSample/second without changing USRP clock?

A) I have been working with fractional rate decimeter/interpolators inside the USRP FPGA. You can put a fractional rate decimeter inside the FPGA which decimates by 6.4. It is a bit hard to get them to fit but you get a lot of flexibility out of them without changing the clocks. .(#FIXME# is this correct ?)

Q) The file usrp_basic.h has the following member:
long fpga_master_clock_freq () const { return 64000000; }
If I want my system to work at 50 MHz, will I need to change this constant?

A) Yes. The idea was that there was a single place in the code (fpga_master_clock_freq) that "knows" the actual frequency.

Q) I've got a USRP v4 board that I'm planning to run off of a 13 MHz external clock. However, I'm intending to use a pair of RFX 900 daughterboards, and I'm wondering if there are any known modifications needed to allow these daughterboards to operate with a 13 MHz clock?

A) You need to go into the db_flexrf.py and db_flexrf_mimo.py files in the gr-usrp/src directory. Any references to 64 MHz need to be changed to 13 MHz. You also need to make sure that you tell the usrp object what its sampling rate is.

Q) What is the voltage level for the USRP external clock? Is 1.0V peak to peak ok? Do I need to add a dc offset to my external clock to prevent the negative swings?

A) Yes 1V p-p for USRP rev 4. Use 3.3V for rev 2 and 3.

Q) What are the limits on the USRP external clock? Will using a “non-standard" clock frequency break any parts of Gnuradio?

A) You can drive the clock line anywhere between about 10MHz and 65MHz. Be sure to edit usrp_basic::fpga_master_clock_freq() to return the new value.

Q) I re-clocked my USRP from 64 to 65.536 MHz and have recently discovered that I & Q samples are no longer orthogonal, rather they seem to be only 70 degrees or so apart. I would guess this is due to some hard coded values in the CIC or HB filters? Is there a USRP command to correct for this phase imbalance?

A) That's an odd problem. Sounds like a timing issue. Keep in mind that the ADCs are only specified to work to 64 MHz and the FPGA image is compiled to meet timing at 64 MHz. You might need to recompile it with the stricter timing numbers. All of the DSP stuff in the FPGA should not care about the absolute frequency as long as it meets timing.

Loading FPGA Bit Stream Questions

Q) How the USRP FPGA configuration bit stream file loaded?

A) When using GNU Radio USRP functions, the host library takes care of loading the ".rbf" file into the FPGA for you. This takes place over the USB.
However, you can manually load the .rbf using the usrper command:

```
$ usrper --help
usage:
  usrper [-v] [-w <which_board>] [-x] ...
  usrper load_standard_bits
  usrper load_firmware <file.ihx>
  usrper load_fpga <file.rbf>
  usrper write_fpga_reg <reg8> <value32>
  usrper set_fpga_reset {on|off}
  usrper set_fpga_tx_enable {on|off}
  usrper set_fpga_rx_enable {on|off}
  ----- diagnostic routines -----
  usrper led0 {on|off}
  usrper led1 {on|off}
  usrper set_hash0 <hex-string>
  usrper get_hash0
  usrper i2c_read i2c_addr len
  usrper i2c_write i2c_addr <hex-string>
  usrper 9862a_write regno value
  usrper 9862b_write regno value
  usrper 9862a_read regno
  usrper 9862b_read regno
```

```
$ usrper load_fpga <name of your .rbf file>
```

will load the .rbf file into the FPGA. **Please note** that it is possible to permanently damage the USRP by loading buggy code into the FPGA.

Q) When we program the FPGA using python programs (eg. usrp_fft.py), are the same .rbf files generated? Where?

A) The files aren't generated, they are loaded. By default, an rbf is loaded from /usr/local/share/usrp/rev{2,4}. The one used unless you specify the fpga_filename constructor argument when instantiating a usrp source or sink is std_2rxhb_2tx.rbf. FYI, we load std.ihx, standard firmware for the Cypress FX2 USB controller, before loading the FPGA .rbf file. We load the FX2 firmware over the USB. That firmware implements additional control endpoint commands, including those that know how to load an .rbf into the FPGA.

Timing Latency Questions

Q) We can calculate the latency introduced by the USB since the USB packet is only sent when we have a sufficient amount of data collected in the USRP buffer, i.e., the smallest allowed USB packet is 512 byte, and the largest one is specified by the user by two parameters, *fusb_nblock* and *fusb_block_size*. As a paper mentions: (http://nesl.ee.ucla.edu/document/show/242), the USB delay (Tu) is according to the equation:

Tu = f(512, fusb_block_size*fusb_nblock) / (fs*sample_size)

Where:
f(x,y) is depends on the data in the buffer and it is at least x and at most y and *fs* is the sampling frequency. Since we use complex 16 bit samples, the sample size is
*sample_ size = 2 *16bit= 4 bytes.*
So, does that mean if we have large product of fusb_block_size and fusb_nblock, then the theoretical maximum delay will be increased?

A) Yes that is true. If you're trying to minimize latency you want the smallest values that work reliably (no over/underruns) and with acceptable overhead. If you enable real time scheduling, you can reliably use smaller values. Try fusb_block_size 2048 and fusb_nblock 4 or 8. You may be able to run with fusb_block_size 1024. It depends on your data rate across the USB.

Q) What is the theoretical minimum roundtrip latency that we can achieve with the USRP?

A) Different cases:
1) On the host I think that the minimum number of packet buffers we can live with is 2 in each direction. This is to ensure that the EHCI USB host controller end point queue always has something in it for our end points. This is required to keep the throughput up and to keep us from suffering underrun or overruns on the USRP.

2) On the FX2 the minimum we can use is two in each direction.

3) On the output path to the FPGA, there's a FIFO, but the pipeline runs if there's any data in the FIFO. This doesn't appear to add any delay. In USRP, we're currently running with 512 byte packets.

Theoretical Calculations:
Let the PKTSIZE = 512

In both directions we can't start the transport until you've assembled a full packet. Assume we're running at 32MB/sec, t0 = 16usec

The transport time of a single packet from the host controller to the FX2 is t1 = PKTSIZE (bytes)/32(Mbyte/sec) = 16 usec

The transport time of a single packet from the FX2 to the FPGA is about

t2 = 4 us + PKTSIZE (bytes)/96(Mbyte/sec) = 9.33 usec

[The 4 us is firmware overhead in the FX2. It could be improved]

The best case:
Outbound looks like = t0 + t1 + t2 = 41.33 usec

Inbound is the same 41.33 usec.

With 512 byte packets, total round trip latency

2 * (t0 + t1 + t2) = 83 usec

Lowering the packet size to 64 bytes would reduce this to

2 * (2 + 2 + 4.7) = 8.7 us
[This is optimistic, since our throughput will drop, and there are lots of other unaccounted factors. E.g., greater USB overhead because of the small packets]

Q) How we can practically measure the round trip latency?

A) Here's an easy way to measure round trip latency:
Hook up a signal generator to one of the Rx basic inputs and to one input of an oscilloscope. Hook up one Tx basic output to another input on the o'scope. Write some code on the host that reads from usrp.source_c and writes to usrp.sink_c Look at the delay between the two traces on your o'scope.

USB Controller FX2 Questions

Q) Am I correct in saying that the usrp code treats the device as a stream of data entering the system via the USB connection?

A) It's a little more complicated than that, but that's the basic idea. USB supports three kinds of transfers across the USB: command, bulk read/write and isochronous. We use command packets to configure the USRP, load firmware, the FPGA bitstream, etc. We use endpoint 0 for that interface. In addition, we use two other endpoints, one for streaming input data and the other for streaming output data. These endpoints are used with bulk transfers, since they are the USB transfers that support the highest throughput. I suggest reading chapter 9, "USB Device Framework" of the USB 2.0 specification. http://www.usb.org/developers/docs/ It covers most of what you need to know to build a USB peripheral.

Q) How many endpoints we can support with FX2?

A) The FX2 is limited in the number of endpoints that it supports. See page 15 of the CY7C68013 datasheet. Right now we are running with endpoints 2 and 6 "quad buffered", 4 512 byte buffers each. One is used for Tx data, the other for Rx data. It would be possible with a bit of hacking to run endpoints 2, 4, 6 and 8 "double buffered", 2 512 byte buffers each. That would give us 4 bulk endpoints to work with (and would also lower the worst case latency).

Q) I am looking at the tx_buffer module. From my understanding, that module does the interlacing of the data to be transmitted (2 I channels and 2 Q channels). Is that correct?

A) It implements the transmit direction part of the FX2/FPGA GPIF interface. It also demuxes the data to be transmitted and sends it to the appropriate DACs.

Q) Does FX2 reformat incoming/outgoing FPGA data?

A) No. The formatting of the data is all handled in the FPGA. The FX2 firmware doesn't even see the bulk data going in and out. The data is transferred by DMA to/from the GPIF from/to the FPGA USB buffers.

Q) Also, next to the bus_reset input declaration there is a comment saying "Used here for the 257 hack to fix the FX2 bug". Which bug is it talking about?

A) On the FX2, the WR pulse is asserted one clock longer than it should be.

Q) I think that along the way I misunderstood the purpose of the write_count register. How does it actually work? WR triggers every time a 16 bit packet is ready from the FX2 doesn't it?

A) write_count counts from 0 to 256, then back to 0. It's at 256 when WR is still asserted but there's really no data to receive. This works around some strange behavior in the FX2 GPIF interface and/or programming.

Q) The wreq trigger of the FIFO is triggered by (WR & ~write_count[8]). Does this mean that only 256 16 bit samples enter the FIFO before the WR is removed? Why is this? How could I determine exactly when there is I or Q sample that must be written into the FIFO?

A) wrreq tells the FIFO when data should be written to the FIFO. So, we write when (WR & ~write_count[8]). That is, when WR is asserted, but the count does not have 0x100 bit set. As I recall, WR is asserted an extra cycle, and the counter trick works around this.
The block called tx_fifo is one of their standard blocks. It's the dual clock version of the fifo, and is used (amongst other things), to bridge between the USB clock domain (.wrclk(usbclk)) and the signal processing clock (.rdclk(txclk)).

Q) When the FX2 detects the have_space pin on the FPGA, does it transfer 1 entire buffered USB packet to the FPGA, then re-check the have_space pin, right?

A) Yes. Have_space is used by the feeding FX2 to know if the FPGA can handle one more packet. The have_space pin is used by the FX2 to know if the FPGA FIFO can store AT LEAST one more packet.

Q) Would it be reasonable to assume a 1 clock delay between the last byte of one 512-byte packet being written to the FPGA and the first byte of a second 512-byte packet being written to the FPGA?

A) Yes. That shouldn't be a problem. There's software inside the FX2 that polls the pin. You've got at least 100 ns between packets, probably more.

Q) I couldn't find any documentation on what is stored in the USRP motherboard EEPROM?

A) On the motherboard, we're currently using a "C2" formatted EEPROM so that we can run code at power-up time to get the 9862's into a reasonably low-power state. The details of the C0 vs C2 FX2 boot can be found in the FX2 Technical Reference Manual.

http://www.keil.com/dd/docs/datashts/cypress/fx2_trm.pdf

The EEPROM on the motherboard stores some USB info, and some very simple code to put the board in a low power state when it powers up. It also blinks one LED quickly. Once you start running anything on the USRP, a more complete firmware (which is much bigger than the EEPROM would hold) is sent over the USB bus.

Q) What about the USRP operation of Cypress EZ-USB FX2 USB Microcontroller ?

A) Many points:
1- Official documentation:
http://www-corot.obspm.fr/COROT-ETC/Files/CY7C68013.pdf
2- The main controlling program is usrp_main.c (compiled firmware for the 8051 in the FX2 USB chip by SDCC) found in gnuradio/usrp/firmware/src/usrp2
3- The following loops were examined from the usrp_main.c program:

```
main( )
{
Initialize USRP
Initialize GPIF
Patch USB Descriptors (read from EEPROM, set HW device ID)
Setup Autovectors
Install USB Handlers
Re-enumerate
Run main_loop( )
}

main_loop( )
{
Check for USB setup packets
Check and log the RX Overruns and the TX Underruns
Check for packets to send back to host
Check for packets to send to FPGA
}
```

4- USB Endpoints:
Different USB endpoints (EP) are used to logically separate different operations occurring on the bus into separate flows. There are currently 3 USB endpoints checked/used within the main_loop() as described earlier.

Endpoint	Description
0	Control/status
2	Host -> FPGA
6	FPGA -> Host

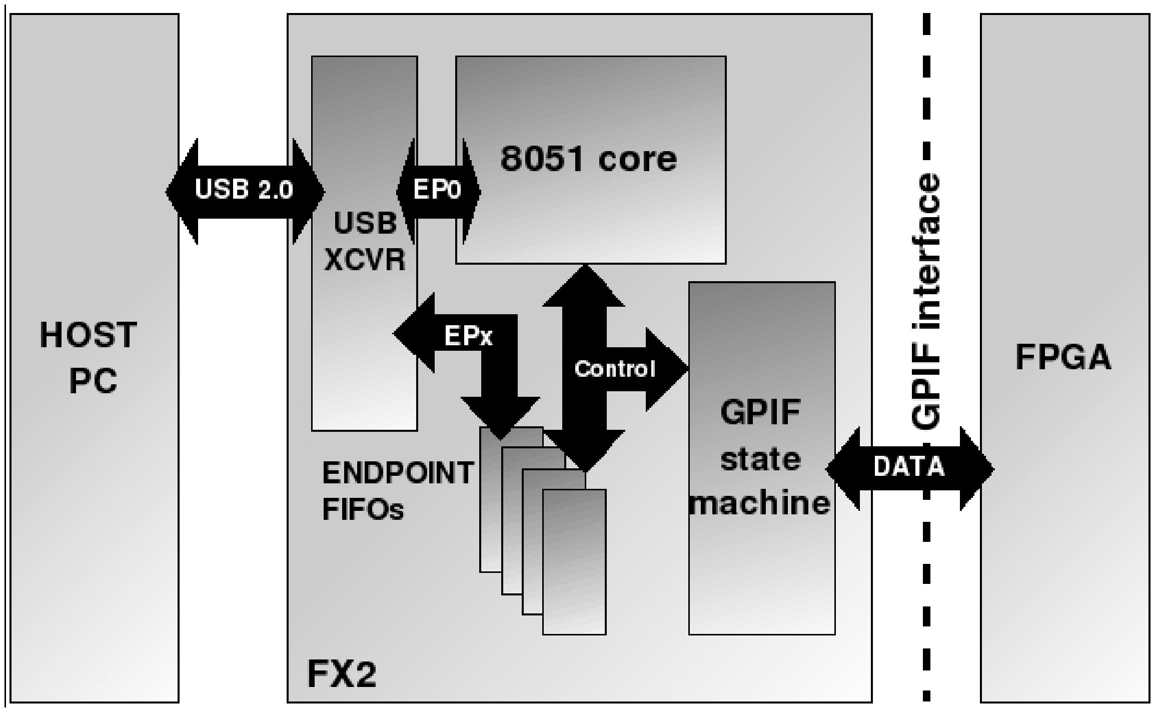
5- USB Transfers are always 512-byte bulk transfers. All control information is written using endpoint 0 and the vendor commands. These commands are separated into two different

categories: VRT_VENDOR_IN and VRT_VENDOR_OUT. These are processed in the app_vendor_cmd( ) function seen in usrp_main.c.
All control communication between the FPGA and FX2 microcontroller is done over the SPI.
Daughterboards all seem to be controlled by the generic VRQ_I2C_* and VRQ_SPI_* commands.
6 - VRT_VENDOR_IN Commands:
VRQ_GET_STATUS
GS_TX_UNDERRUN
GS_RX_OVERRUN
VRQ_I2C_READ
VRQ_SPI_READ

7 -VRT_VENDOR_OUT Commands:
VRQ_SET_LED
VRQ_FPGA_LOAD
FL_BEGIN
FL_XFER
FL_END
VRQ_FPGA_SET_RESET
VRQ_FPGA_SET_TX_ENABLE
VRQ_FPGA_SET_RX_ENABLE
VRQ_FPGA_SET_TX_RESET
VRQ_FPGA_SET_RX_RESET
VRQ_I2C_WRITE
VRQ_SPI_WRITE

Q) How the USRP FX2 works?

A) The FX2 microcontroller contains an embedded USB 2.0 transceiver and handles all USB transfers with the upstream USB host. It presents a data bus to the outside world (in this case the FPGA), with generic control signals which can be programmed to behave in a custom manner. This interface is called the GPIF (General Purpose Interface),
The FX2 also handles all USB control requests (via *endpoint 0*), which all USB-enabled devices must support to fully comply with the USB standard. These include responses to device capability interrogations and standard setup requests.



FX2 Internal Block Diagram (From Kalen Thesis)

A simplified view of the FX2 is shown in Figure above. It houses an industry-standard 8051 microcontroller core (with several extensions and enhancements), which handles all internal control. The 8051 initializes the transceiver, which handles the actual USB transactions. It is also responsible for configuring the FX2's general-purpose I/O ports (not shown) and the GPIF state machine. Data is transferred in a USB system via endpoints, which are similar to *Ethernet* network socket. Each endpoint must have a specified data-flow direction, either IN or OUT (USB-centric), except the control *endpoint 0* which is bidirectional. The endpoints, must also have defined data transaction types which indicate their bandwidth requirements.
Data entering or leaving the FX2 on the USB host side is stored in the endpoint FIFO's, which can be configured to have various sizes and levels of buffering. The GPIF has direct access to these FIFO's, allowing for seamless data transfer between an external device and the USB host through a series of buffered FIFO's.

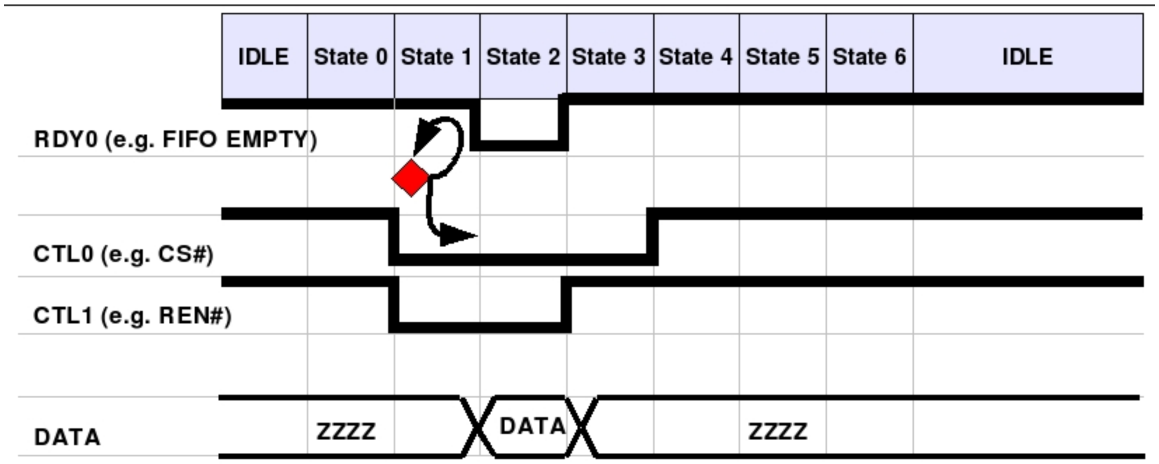
Q) What about FX2 firmware?

A) As well as general house-keeping and configuration, the FX2 firmware is responsible for the following areas:
• GPIF initialization
Rather than attempt to handle data transfers directly at USB 2.0 high-speed (480MBits/s), data transfers are carried out by the GPIF. The FX2's 8051 core initializes this interface.
• USB control request handling
The 8051 core responds to control requests sent by the USB host over *endpoint 0*. All USB compliant devices must return responses in a pre-determined format to standard requests about the device (e.g. *GetDeviceName* which returns the device's description).
• USB transfer requests

These are implemented as a polling loop that loads GPIF setup registers with the transfer parameters as and when requests are received from the host. The FX2 firmware is written in *C*, and compiled with the open-source compiler, *SDCC* (Small Device Cross Compiler). The embedded functions are invoked using the *usrp_basic* and *ursp_prims* libraries.

Q) What is the FX2 GPIF?

A) The GPIF (General Purpose Interface) is a mechanism implemented by the FX2 to allow for simple interfacing with many different types of devices. Essentially it is a bi-directional data bus with a set of generic control signals which are governed by a configurable state machine. The GPIF can be configured to be the interface bus master or can be driven as a slave by the device itself. The GPIF is the bus master. Six states can be defined for a particular waveform, with decision points that dictate the state transitions depending on the values of the generic control lines.
There are four bus cycle waveforms available which can be configured. These are:-
• Single Write
This bus cycle writes a single data word from the USB to the device (all transactions are defined to be USB-centric)
• Single Read
This cycle reads a single word from the device.
• FIFO Write
This cycle writes a block of data based on some previously setup parameters. During the transfer state, data flow can be throttled by the receiving device.
• FIFO Read
This cycle reads a block of data from the device, and is used to stream ADC data up to the host.
An example of a simple GPIF FIFO read waveform is shown in Figure below.



GPIF waveform Example (From Kalen Thesis)

The GPIF has six control inputs *RDY5:0*, and six control outputs *CTL5:0*. These can be used to implement a great variety of standard and proprietary bus control cycles. In the figure above, the decision point in state 1 will hold the cycle in a stalled state until *RDY0* (in this case representing

FIFO Empty Flag) is de-asserted when the GPIF state machine will advance to state 2 and latch in the data presented on the *DATA* bus. The FX2 implements the USB endpoints as internal endpoints *EP0, EP1, EP2, EP4, EP6* and *EP8*, which can be configured in various ways to suit the application's buffering requirements. *EP0* is always a 64 byte *CONTROL* endpoint and *EP1* may only be of the *BULK* or *INTERRUPT* types (also 64 bytes). The remaining endpoint FIFO's can be configured to be double-, triple- or quad-buffered, with a maximum total buffer size of 4K. In this manner, data can be continually streamed to or from the host, provided adequate flow control is in place.
Cypress Semiconductor has developed *GPIF Designer*, a freely available application which can be used to design GPIF state machines with a graphical interface. The application allows a user to customize the state machines, after which the appropriate configuration and initialization source code is generated which can be included with the main FX2 firmware. This simplifies the process of configuring the GPIF waveform registers which would otherwise have to be coded manually. In the USRP's case, the developers chose to parse the *GPIF Designer's* output code, and to insert their own initialization routines. This was done in part because the output code generated by *GPIFDesigner* is somewhat ambiguous (bugs have been encountered by the *GnuRadio* team.

Internally in USRP, the GPIF acts autonomously from the actual FX2 8051 core and provides data to and from the internal endpoint FIFO's for streaming up to and down from the USB host. The USRP FX2 interface at the application level is defined and implemented in the *usrp_prims* and *usrp_basic* source and header files.
The Usrp_prims defines a set of functions implemented on the FX2 in response to control requests via USB *endpoint 0* that extend the generic USB function set. These function primitives typically write to or read from various FX2 registers, or initiate data transfers via the GPIF.

The Usrp_basic provides a simple API to the *usrp_prims* functions in the form of an object-oriented structure. The interface to the system provides higher-level functionality, hiding most of the underlying interface mechanisms.

Q) What is the usrper?

A) It is a data streaming test application. It can load FPGA configuration data (fed serially into the configuration lines of the FPGA bitwise), assert debug USRP LEDs, read and write data to the USRP.

On start up, *usrper* first attempts to create an interface object that can communicate with a valid USRP device. This creates a handle to a generic USB device, and then traverses all the devices found attached to the USB, searching for the FX2's vendor and product ID numbers (*0x04B4* and *0x8613* respectively). On successfully locating a powered, un-configured FX2 device, *usrper* then downloads the *GnuRadio* firmware to the FX2, which sets up the system. After system setup, the firmware loads custom values into the Vendor and Product ID fields and initiates a software reset, causing the FX2 device to disconnect and then reconnect to the USB bus.
This process is called re-enumeration, and the *Linux* kernel now sees the USRP board as a USB device with the Product ID and Vendor ID *0xFFFE* and *0x0002*, which correspond to a configured USRP device developed by the Free Software Foundation.
Once this has been achieved, calls are made to USRP-specific methods which can be reads, writes or control requests.

• Control requests:
These enable or configure various aspects of the USRP system, and are made to USB endpoint 0. The firmware intercepts these calls (interrupts are generated on all USB control requests that are received from the host) and services them.
They form vendor-extension functions, and carry out various data transfers to and from internal registers based on the host's control request parameters.

For example to download the FPGA's firmware, the configuration bit stream data is sent via endpoint 0 with the appropriate control parameters indicating a request to configure the FPGA. The FX2 intercepts each byte in this stream, and sends it bitwise over the FPGA's configuration lines (the Altera EP1C12 FPGA supports serial configuration modes). While servicing the control request, the FX2 firmware sends back the appropriate handshaking signals to the host to indicate that the control request is being handled correctly.
• Read requests:
The USB requests data from the USRP by making read requests over endpoint 2 (set up for 512 byte bulk **IN** transfers). The FX2 intercepts these requests and primes the GPIF transfer state machine with the transfer parameters before relinquishing control to the GPIF.
The GPIF enters the FIFO read state machine and remains in this configuration until the transfer is complete, or an error has occurred. The FX2 handles all acknowledge and handshaking signals with the USB host transparently from the firmware.
• Write requests:
As with read requests, the host PC makes generic USB write requests, this time to endpoint 6 (configured for 512 byte bulk **OUT** transfers). The GPIF in this time set up for the FIFO write state machine, and transfers data until the transaction is complete, or an error is encountered.

Q) What does the FX2 low-level interface library (USRP_PRIMS.CC) do?

A) This library code forms the direct interface with the FX2 as a set of low-level functions. Data transfers and various control requests are passed to the FX2 via USB *endpoint 0*, while the data received is sent back via *endpoint 2*.

Q) What does the FX2 application-level interface library (USRP_BASIC.CC) do?

A) This library presents a higher-level interface to the FX2, in the form of object-oriented methods which call the functions in usrp_prims.cc.

Q) Why we use GPIF mode?

A) We use the FX2 in GPIF mode because it allows us to burst data across the GPIF interface at 96 MB/sec.

Q) USRP-FX2 interface - As I understand it, all control calls to endpoint 0 are handled within the FX2 and do not require the involvement of the FPGA, right?

A) Yes

Q) GPIF / FPGA firmware - How many GPIF waveform configurations do you implement?

A) Two: FIFORd and FIFOWr

Q) Do you handle FPGA configuration explicitly using the FX 8051 core?

A) We bit bang the FPGA configuration from the 8051.

Q) Please could you point me to the correct FX2 source to determine your GPIF CTLx/RDYx lines and waveform descriptions,?

A) We used the Cypress GPIF designer. See gpif.gpf.

Q) How we can modify the USRP which uses USB2.0 (480Mbit/sec) high speed interfaces to use USB 1.1 (12Mbyte/sec) full speed controller. The USB 1.1 sends 64 byte packets, while USB2.0 uses 512 byte packets?

A) If you go to:

http://gnuradio.utah.edu/trac/browser/gnuradio/trunk/usrp/fpga/sdr_lib/tx_buffer.v

and look at line 94, you will see the test for "end of packet". write_count[8] goes high when we have put 256 elements (512 bytes) into the FIFO. You would need to modify this to write_count[5] which will go high when 32 elements (64 bytes) have been put into the FIFO.

You probably also need to modify some firmware. Also you have to decimate the source down to a very low data rate, so it won't overflow.

Q) Doing suggested above modifications, it looks like data is coming out, but it looks like I get 64 bytes out, then there is a "hiccup" about 5 microseconds long. I am getting suspicious the PC doesn't get the data on the USB bus fast enough. I am wondering if there is a way to let the buffers in the FX2 chip fill up more before the FPGA starts pulling data from the FX2?

A) Unless your FX2 code sets up the transfer size to 64, then this is probably what is happening. That is, the host is sending 64 bytes, but the FX2 is ignoring the real length, and is assuming that it's 512 bytes. Remember that the GPIF is currently set up to DMA 256 16-bit values. Perhaps that part needs changing. The magic value is probably buried in the WaveData table in usrp_gpif.c.
Another thing you could try is to set up the GPIF in a non-flowstate mode. You'd need to use the Cypress tool to do this (or Larry Doolittle's perl (?) code. Running in full speed, you don't need to be able to burst data at 96MB/sec between the FX2 and FPGA.
Note:
There is a suggested patch to make USRP able to use USB 1.1 at:
http://lists.gnu.org/archive/html/discuss-gnuradio/2006-04/msg00284.html

Q) I looked over the GPIF stuff with the Cypress tool I do not see any reference to transfer size in there. I am thinking the transfer size is set in the FPGA?

A) Nope. The GPIF is in charge of the transfer. I'm assuming that what you are seeing is that the FX2 is bursting a 256 word transfer, given how you have programmed everything. BTW, I'm not kidding about this being nearly impossible without a logic analyzer. What's currently really happening on the GPIF bus? (Bring out the relevant GPIF pins to the daughterboard debug headers.)

Looking at page 10-16 (10.3.2.2.2 Decision Point States) of the FX2 Technical Reference Manual (and dusting off my memory), the waveform decision point is controlled by the "Transaction Count Expired" signal. Page 10-24, "LOGIC FUNCTION Register", TERMA and/or TERMB will be coded as RDY5 (or Transaction-Count Expiration, if
GPIFREADYCFG.5 = 1 (which it is))
See page 10-41, 10.4.3.1 "Transaction Counter".
To use the Transaction Counter for FIFO "x" load GPIFTCB3:0 with the desired number of transactions. When a FIFO-READ or -WRITE waveform is triggered on that FIFO, the GPIF will transfer the specified number of bytes (or words, if WORDWIDE=1).

See ### HERE ### below (from usrp/firmware/src/usrp2/usrp_main.c):

Are you setting these to 32 instead of 256? If not, that's probably the root of the problem.

```
    // Next see if there are any "OUT" packets waiting for our attention,
    // and if so, if there's room in the FPGA's FIFO for them.

    if (g_tx_enable && !(EP24FIFOFLGS & 0x02)){  // USB end point fifo is not empty...

      if (fpga_has_room_for_packet ()){   // ... and FPGA has room for packet

        GPIFTCB1 = 0x01; SYNCDELAY;        #######  HERE #######
        GPIFTCB0 = 0x00; SYNCDELAY;        #######  HERE #######

        setup_flowstate_write ();
        SYNCDELAY;
        GPIFTRIG = bmGPIF_EP2_START | bmGPIF_WRITE; // start the xfer
        SYNCDELAY;

        while (!(GPIFTRIG & bmGPIF_IDLE)){
          // wait for the transaction to complete
        }
      }
    }

    // See if there are any requests for "IN" packets, and if so
    // whether the FPGA's got any packets for us.

    if (g_rx_enable && !(EP6CS & bmEPFULL)){ // USB end point FIFO is not full...

      if (fpga_has_packet_avail ()){ // ... and FPGA has packet available

        GPIFTCB1 = 0x01; SYNCDELAY;       ####### HERE #######
        GPIFTCB0 = 0x00; SYNCDELAY;       ####### HERE #######

        setup_flowstate_read ();

        SYNCDELAY;
        GPIFTRIG = bmGPIF_EP6_START | bmGPIF_READ; // start the xfer
        SYNCDELAY;

        while (!(GPIFTRIG & bmGPIF_IDLE)){
          // wait for the transaction to complete
        }

        SYNCDELAY;
        INPKTEND = 6; // tell USB we filled buffer (6 is our endpoint num)
      }
    }
```

Unless you get lucky and the modification above labeled "HERE" works, you're _really_ going to want access to a logic analyzer.

Q) I'm looking into writing a native USB driver for Windows and the USRP; I'm trying to find out some information on the USB interface of the USRP.

A) See:
  usrp/firmware/include/usrp_interfaces.h
  usrp/firmware/include/usrp_commands.h

Q) Does anyone know where the best place is to find documentation (if any) of the USB data structures that are sent on interface 0, 1 & 2 (ep0, ep2 and ep6) so that I can achieve this. I assume the ep0 data structures are that of the USB Device Request structure on the default control pipe, but I also assume there are more control requests than the standard USB ones. Another thing I was wondering, since the vendor id is a standard free for all type, how do you actually determine if it is indeed a USRP that you are connected to, and on top of that if you are connected to an FX2 chip which is not the USRP and has not been initialized, can you determine that it is an FX2 that in on a USRP or not?

A) All the shipped USRPs have a non-Cypress USB VID/PID burned into them. There is no confusion between a USRP and an un-programmed FX2. We use: VID = 0xfffe, PID = 0x0002. We distinguish board revs and whether or not we've loaded our firmware via the DID. See usrp/firmware/include/usrp_ids.h
Pretty much every question about how to control the USRP is answered in one of the .h files in usrp/firmware/include
In particular; see fpga_regs_common.h and fpga_regs_standard.h for the definitions of the configuration registers in the FPGA.

Q) My problem is that I completely miss the low level USB programming skills and I hoped to find some sort of driver for the cypress chip in Linux.

A) No need for any driver programming. Just use libusb.
See http://libusb.sf.net

Q) How do I ensure that the appropriate usb driver(s) are associated with my USB device when I connect to the system?

A) Here is the outline of a few steps:
1. Assign yourself a USB product ID in the space defined by:
firmware/include/usrp_ids.h and get the USRP developers to acknowledge it.
2. Hack host/lib/usrp_prims.cc to accept your PID in addition to the normal USRP ones.
3. Program the FX2 EEPROM to your product ID using usrper i2c_write.
4. Add a usermap file and initialization script to /etc/hotplug/usb/

Q) Is it feasible to modify existing FX2 firmware to my own needs for testing purposes?

A) Absolutely. You can use a slightly modified usrper and test_usrp_standard_rx to get yourself going until you build your own customized executable from the libraries.

Q) We created a custom board based on the USRP. I'd like to avoid recompiling the FX (8051) source code if possible. Is there a function that will allow me to write directly to one of the FX registers? I want to change the frequency of the CLK output from the default of 12 MHz to 24 MHz.

A) No command. In USRP we run the FX2 at 48 MHz after our firmware is loaded.

Q) It was said that USRP FPGA FIFO Buffers is 2K lines. Does it mean 2048 bits?

A) No, it means 2K lines, each line is 32 bits. The FX2 also implements quad buffering in both TX and RX directions, each buffer is 512 bytes.

Q) What do the parameters fusb_nblock and fusb_block_size exactly do in how data is transmitted over the USB to the USRP?

A) The fusb_block_size is the size in bytes of the maximum transfer that we will ask the kernel to make to/from user-space. The fusb_nblock is the maximum number of transfers (of maximum size fusb_block_size) that we can have in flight at any given time. Take a look at fusb_linux.{h,cc} for the details.

Q) If there is less data available on the USRP (i.e. not fusb_nblock*fusb_block_size), does it still get sent over the USRP to the computer?

A) Yes. The USRP packages data into 512 byte USB packets and sends them as soon as it can. That's 128 complex samples (16-bit I & Q).

Q) How is the smallest amount of data necessary in the USRP defined such that a packet is sent over the USB to the computer? Is it fusb_block_size?

A) Its 512 bytes. It's set in the FX2 firmware.

Q) looking through the FX2 data sheet, the GPIF designer app's generated code and your edit_gpif script output, I (hope) I have an understanding of the actual data flow bus cycle (I am only focusing on FIFORd): You use flow states. There is a flowstate in state S1, which constantly asserts REN and OE (and BOGUS) while the transaction has not yet expired. It de-asserts all CTLx signals when the transfer is complete.
USRP do data transfers on both edges of IFCLK. The FLOWSTB register indicates that CTL4 is your master strobe. I thought CTL4 was a reset signal (CLRST). To summarize:
OE and REN are active-high, and data is clocked out on both edges of IFCLK. Is this correct?

A) Yes, modulo problem with signals being asserted one cycle too long on WR

Q) What part do the RDYx pins play in the flowstate?

A) None.

Q) Do they only affect transitions between S1, S2..S6?

A) Nope, not that either...

I Hope the document was Useful……….Firas