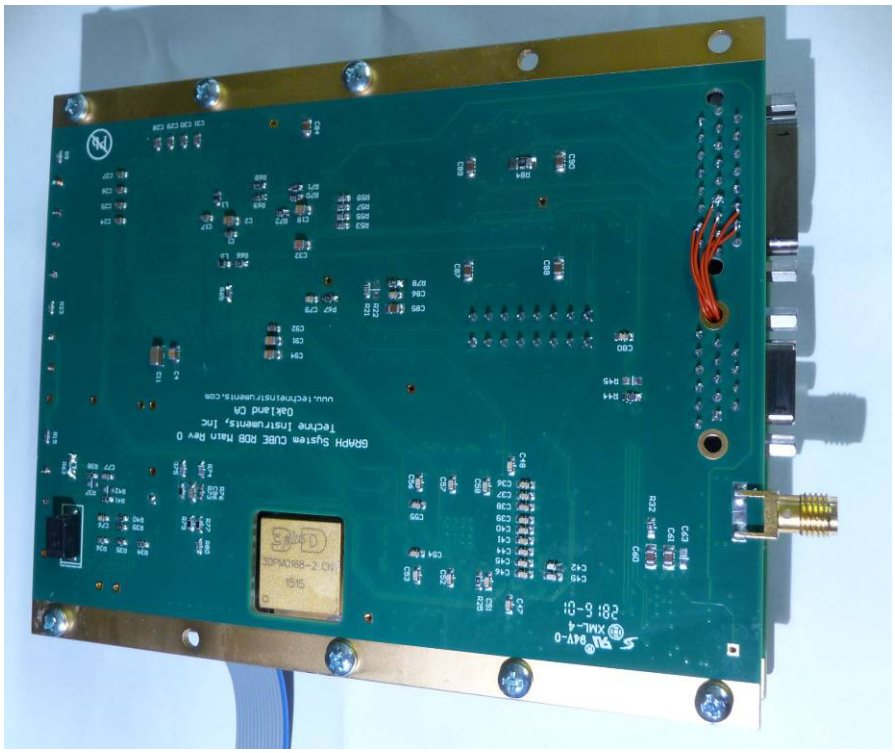
Cubesat RDB Test Report

Rick Raffanti
www.techneinstruments.com

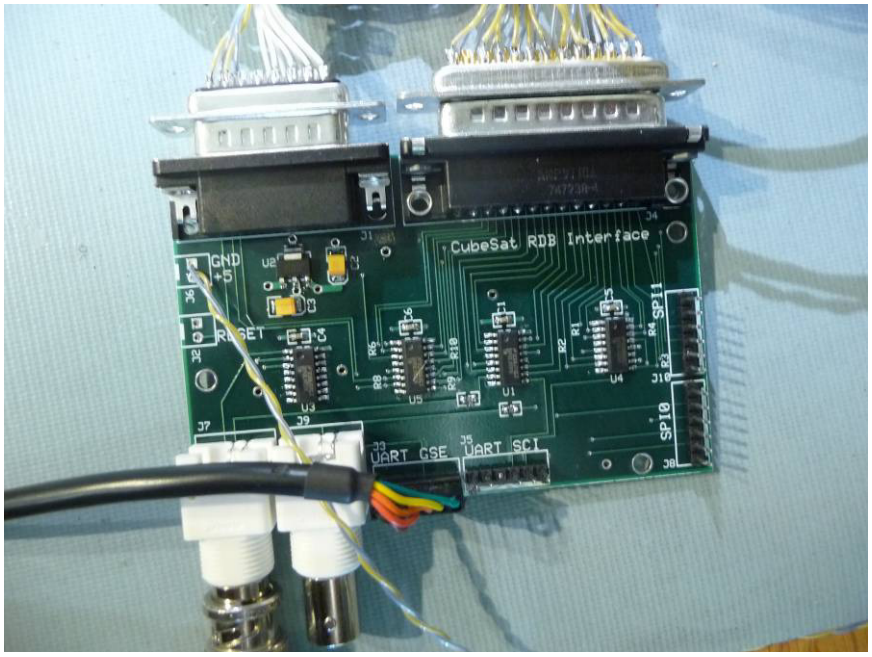
October 16, 2016

I've tested and shipped two prototype systems.

Power board is at the left, main board at the right. The power board folds over; the tall module is the "Latchup Current Limiter" (LCL) which extends a little bit through the square hole when the boards are separated by 0.5" spacers.



RF enters on the SMA, the two MDM connectors connect to a “GSE” board which has some level translators for the LVDS, and connections to two serial ports and two SPI interfaces:



The BNCs on the GSE board are used to bring in the 1PPS and VFC signals. The UART connections are designed to connect to an FTDI USB-to-serial adapter:

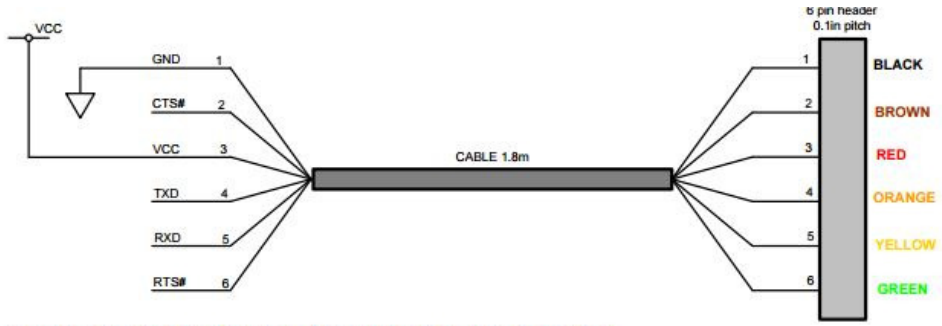
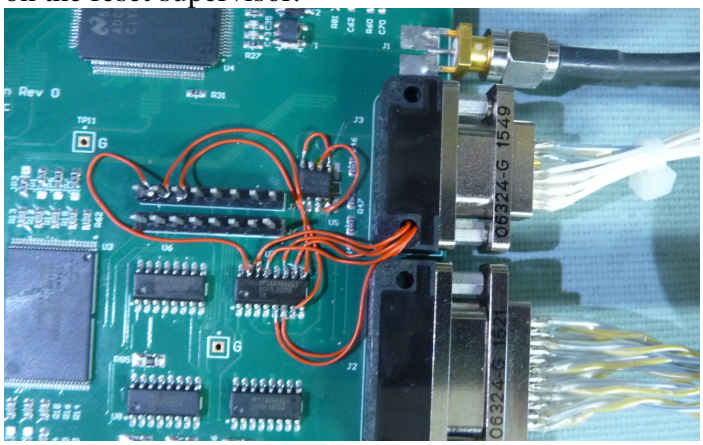


Figure 4.1 TTL-232R-5V and TTL-232R-3V3, 6 Way Header Pin Out

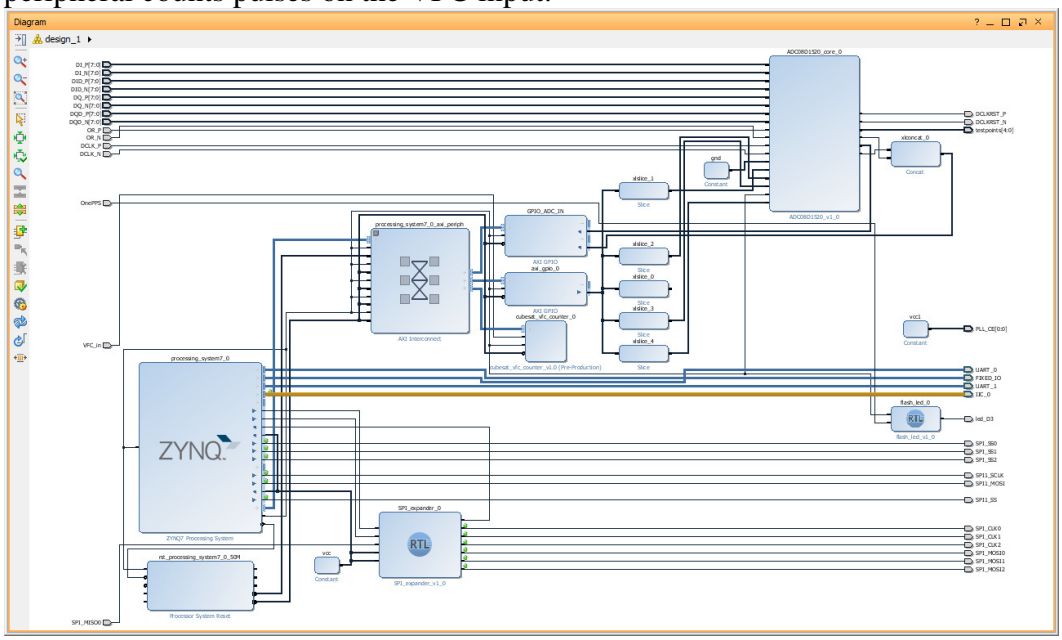
The SPI connections at right are for probing, or connection to peripherals. +5v for the whole system comes in on the twisted pair.

The red wires are the addition of the second SPI link, as well as to correct a pinout error on the reset supervisor:



Firmware

I made a Zynq design with some standard peripherals and a custom IP block to receive the data from the ADC. The standard periperals include 2 UARTs, one I've called "Science" and the other "GSE", two SPI interfaces (one with three SlaveSelect lines, for the PLL chip, the ADC chip, and one of the external SPIs, the other for the second external SPI), and one IIC interface (for the temperature sensor). Another custom peripheral counts pulses on the VFC input.

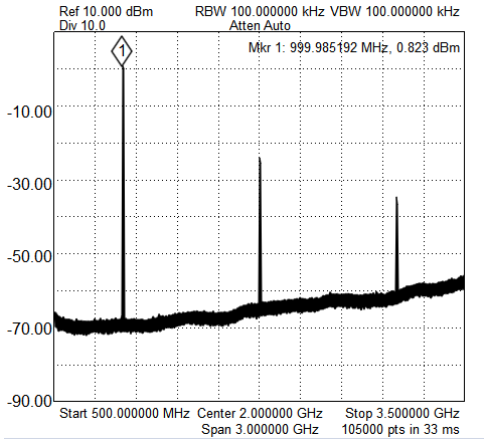


I've written a python 3.x script, "CubeSat_SCI_IF.py" to communicate with the system via the "Science" UART: commands can be sent (for writing registers, etc) and data can be dumped. The "GSE" UART is used to received a command and respond with a basic housekeeping packet. A separate python script, "CubeSat_GSE_IF.py" is used.

```
C:\rix\techne\JPL CubeSat\python>python3 CubeSat_SCI_IF.py
CubeSat RDB Control
Enter
    "P" to setup PLL,
    "A" to setup ADC,
    "" to write a single SPI register,
    "R" to pulse the DCLK_RST pair,
    "M" to reset the MMCM,
    "I" to reset the IDDRs,
    anything else to take a data file,
    or "q" to quit
```

```
C:\rix\techne\JPL CubeSat\python>python3 CubeSat_GSE_IF.py
CubeSat RDB Control
Serial port used: COM4
Enter
    "H" to report Housekeeping,
    or "q" to quitH
VFC Counter =0
Board Temp =36.9375
FPGA Die Temp =44.5916748046875
Enter
    "H" to report Housekeeping,
    or "q" to quit
```

Basic Functionality:
After power-up, the PLL must be set up, using a sequence of writes to the on-chip registers; this is done by typing "P" in response to the prompt.

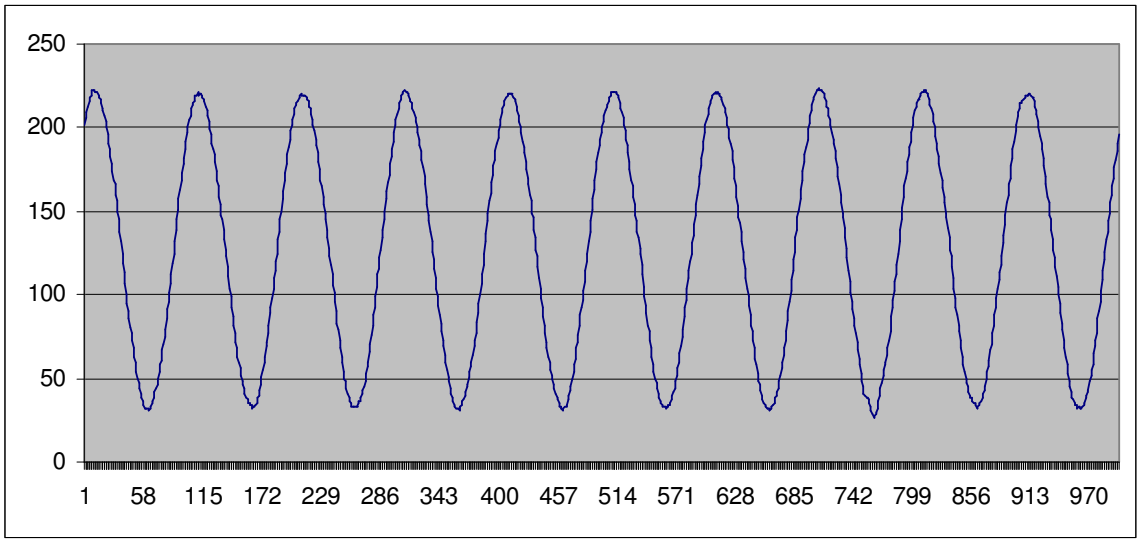


PLL output (ADC clock input). About +1dBm at 1GHz into 50ohm spectrum analyzer

Similarly, the ADC needs to be set up by typing "A". A series of register writes sets the proper data multiplexing, etc; single register writes can also be used to set gain and offset of the two interleaved converter cores.
Typing any other character causes a 1000-sample data series to be stored in C:\Data\CubeSat_out.csv.

The firmware core simply acquires a 1k-by-64 bit buffer of data, 8k 8-bit samples in all, and signals to the processor that data is ready. The processor then sends the bytes via the SCI UART. (This UART is operating at 115.2 kbaud, though it is supposed to be 460.8; my USB cable is limited to 115.2. The baud rate can be set in the Zynq software).

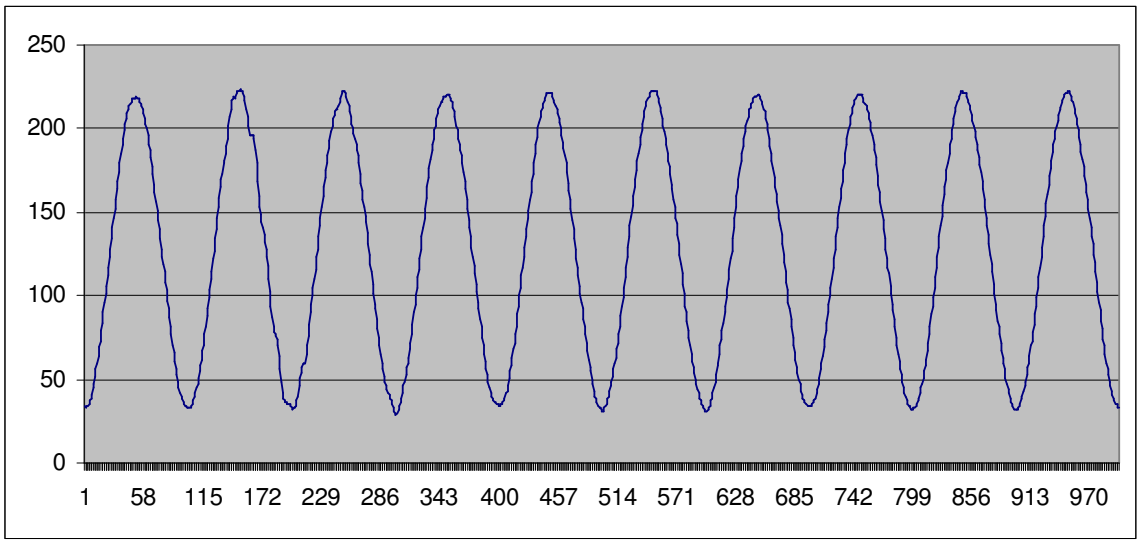
A typical snapshot of 1k samples of a 20MHz 700mvpp input:



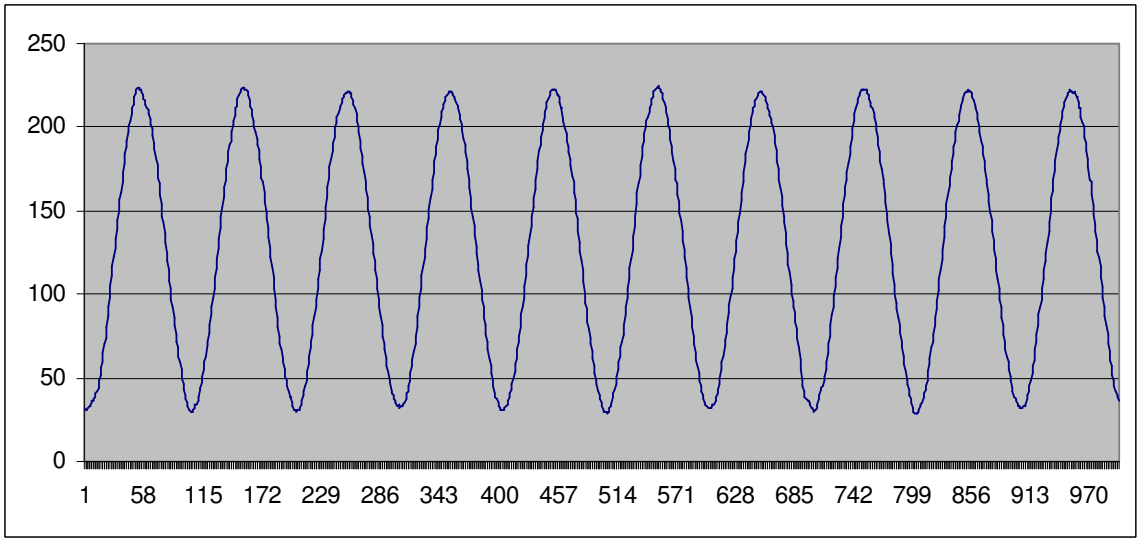
Power draw is 1.48A at 5.0v.

Data Transfer Robustness

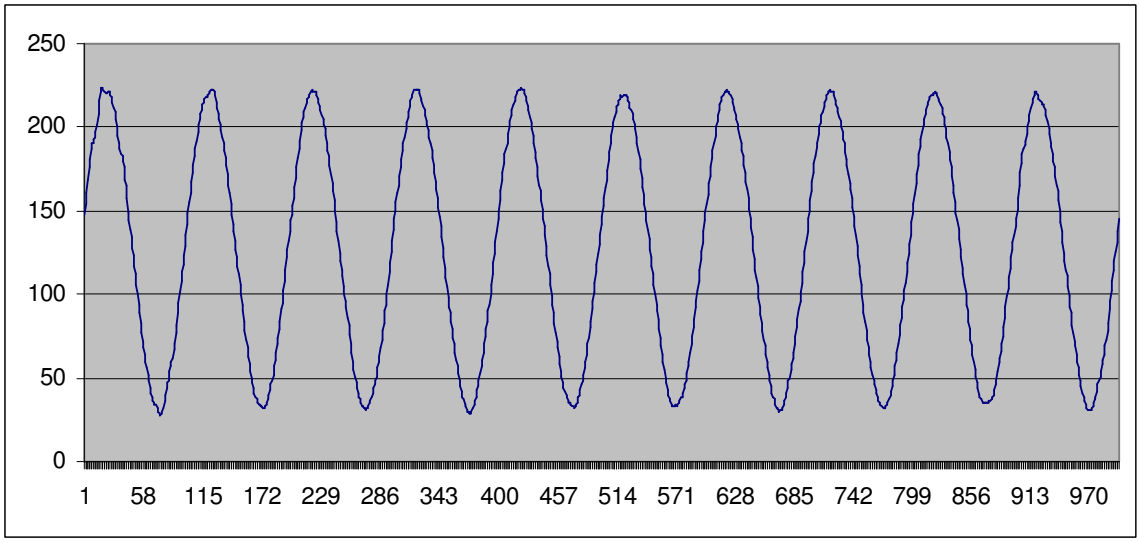
The ADC provides a source-synchronous 250MHz data clock along with the 32 data lanes, each of which transfers data at 500Mb/s. The clock transitions at the same time as the data (although the ADC provides the option of setting the clock to transition in the middle of the data eye, register-selectable). An MMCM in the Zynq programmable logic generates a 90 degree phase-shifted version of the data clock to register the incoming data into the IDDR registers. Due to the potential variations in FPGA timing over process, voltage, and temperature, there is no fixed phase shift setting that can meet the timing constraints (the data eye, with zero transition time, is only 2ns; the variation of delays over PVT is something like 3 ns). To try to demonstrate timing margin I generated bit files for MMCM delays of 0, 45, 135, and 180 degrees as well (you can do this without a recompile by editing the FPGA manually before generating the bitstream). Each step is 0.5ns; all seemed to transfer data cleanly:



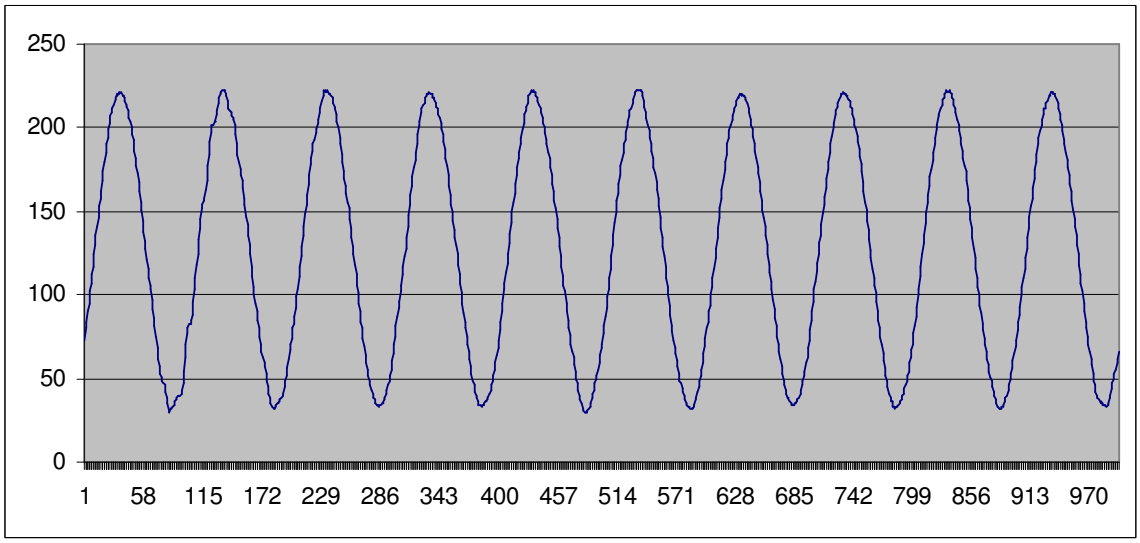
0 degrees



45 degrees



135 degrees



180 degrees

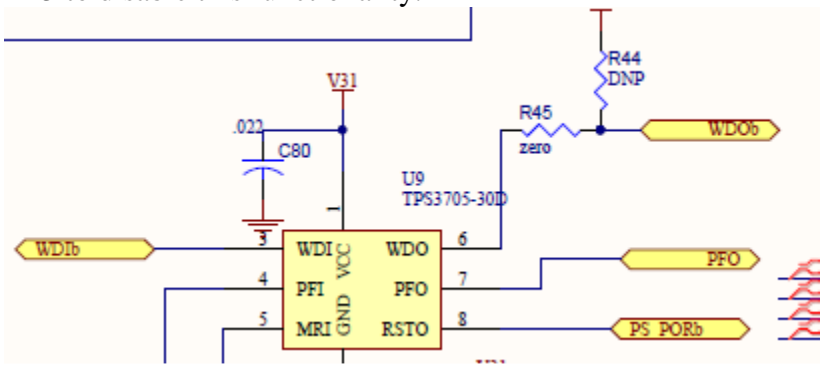
I think this was a valid test, though I might have been kidding myself somehow. This should be looked into more closely, I think, with finer stepping. It could be that the transition times of the signals are so short that I missed the noisy zones. It may be that dynamic control of the MMCM phase will be necessary.

Peripherals

The temperature sensor on the board and the one in the FPGA are both read out successfully from the GSE UART. The ADC has a temperature diode in it that I connected to the XADC (FPGA internal ADC); so far this seems to read out 0 always, although the voltage is about 700mv. I think I perhaps haven't found the right way to access that channel. The XADC provides the FPGA die temperature, and this is working.

Latchup Current Limiter
The LCL from 3D-Plus monitors the current drawn by the FPGA's VCCAUX (1.8v); in the event of an SEU, this current may increase and stay there. The LCL senses the current; when it exceeds a resistor-set threshold, it interrupts the current and signals a fault. I measured the VCCAUX current at 196mA for the system when up and running, and set the LCL threshold at 300mA. But in this configuration, the system would not start up. The Zynq requires more startup current than this (the datasheet seems to say it requires about 600mA during startup). I found that I had to set the threshold to about 1.3A to get it to start up. The LCL has two thresholds controlled by a pin; these are designated "RUN" and "STANDBY". We could perhaps set this signal to RUN during startup and STANDBY thereafter; maybe this could be controlled by a proper pull-up on the control line during configuration, which would then be driven low. When we know the VAUX current draw of the flight design we can consider this more carefully.

Reset Supervisor/Watchdog
After fixing my pinout error on this chip, it provides the proper POR signal which brackets the rise-time of all of the supplies. The watchdog part of the chip, which senses a transition (either direction) on its input and asserts an output if such a transition has not been sensed in about 1.6 sec, also works, but prevents the startup of the system (because no transitions are being generated during power-up and configuration the reset is asserted, preventing the system from starting up). I'm not sure what to do with this. I removed R45 to disable this functionality.



Configuration Flash
The NAND Flash chip itself (128M * 8 bits, 1Gb) works; I've built and run a Xilinx-provided program "xnandpsexample" and can see that data is written and read back correctly. Also, I've compiled and run the necessary "First Stage Boot Loader" program, whose job it is to read the flash on boot-up and copy both the Programmable Logic configuration file to the PL and the application executable to execution memory (in our case that would be the 192k On-Chip Memory). But I have not been able to program the flash itself. Therefore the FSBL runs, but can't do its job, because there's nothing in the flash.

There are two utilities that are usually used to program flash devices: the "program_hw_cfgmem" tcl command available in Vivado, and the "program_flash" utility available through SDK. When I run either of these I receive an error "Problem in running uboot". Uboot is, I think, a second-stage bootloader typically used to load much

larger software images, eg, Linux. I think that both of these programs require a Zynq system with some reasonable amount of processor memory- the image being loaded might be many GB, and I can imagine that one would want a buffer of at least many MB. In any event neither of these runs on our system (I've successfully programmed the QSPI on a zedboard using these).

Since I can write the flash under program control it might be possible to just write the boot image byte-by-byte (to write our own "program_flash") but I suspect this might be a big project- the NAND flash has a whole system for managing bad blocks that has to be understood and managed. But I realize this brings up a bigger issue.

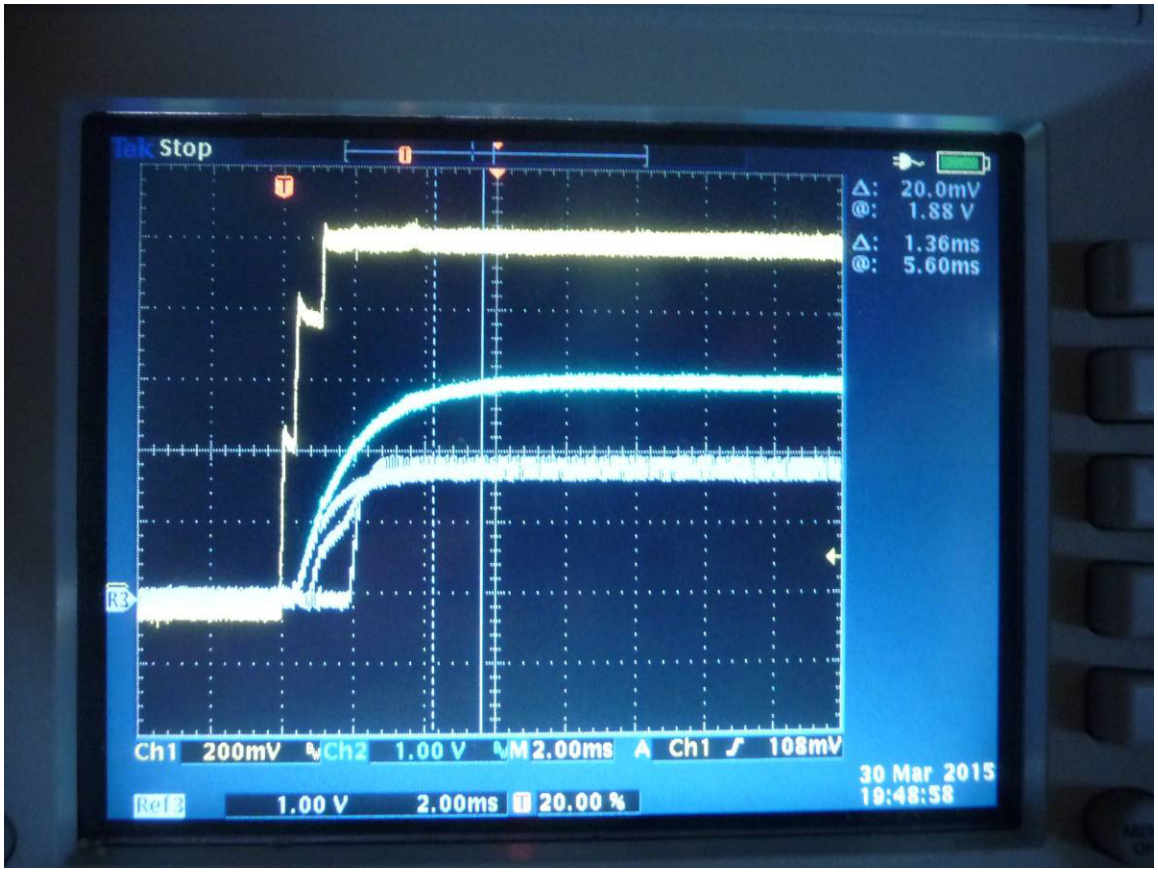
The FSBL performs some necessary steps. For instance, it includes a call to the ps7_init() function which sets up the Zynq processor clocks, IO, etc. Also, it needs to ascertain what sort of boot device is being used (by reading the boot jumpers) then read some registers in the NAND device to determine other operating parameters (size, speed, etc) of the particular device, and it must read the configuration bitstream and write it to the PL via the PCAP (Processor Configuration Access Port). As it stands, my FSBL occupies about 156kB of the total available 192kB of OCM. It should be possible to reduce this (for instance, since we only have to access one type of boot device we can hard-code this and eliminate the reading of this information from the registers), but I don't know how far.

So I wonder what sort of program we anticipate running in the Zynq processor? If it's only servicing the two UARTs, as my test loop does, it won't require a lot of memory. But I think part of the whole purpose of the Zynq system was to do some sort of Single Event Mitigation, where the processor is continually reading the configuration logic and updating it, when necessary, by comparing with an image read from the flash. This might require some complexity.

So, to sum up, I'm stuck on programming the flash, due to lack of DDR. And I wonder if we solve that problem will we have enough program store to run whatever code is necessary. I note that the CHREC Space Processor board has 512MB of commercial DDR.

Power Sequencing

The Zynq has specific sequencing recommendations; the VCCINT (1.0) should come up first, followed by VCCAUX (1.8), then VCCO. These are shown in the scope photo below; the PORb signal, not shown, is asserted low during the ramp time and released about 200ms later.



VCC10 yellow, VCC18, VCC18AUX, VCC19, VCC31

Migration to a flight design
A few PCB design issues need to be fixed: the two pinout errors mentioned above, and the SMA footprint, which required some adjustment.

The NAND boot issue needs to be resolved, either by figuring out how to program the flash without DDR memory, or by adding some DDR.

Power Converters need to be replaced with rad-hard equivalents, which will require PCB design changes as the footprints are different.
The LTM4619IV#PBF 1.0v switching converter could be replaced with the PE99153 6A converter from e2v.
The LT1959 switching converter used to make 1.9v can be replaced with the MSK5048RH, which incorporates the RH1959 die, diodes and power inductor.
The two LT3083 LDO regulators, making 3.1v and 1.8v, can be replaced with the MSK5986RH, which incorporates the RH3083 die.

The Reset Supervisor needs to be replaced. The prototype part used, TPS-3705-30, is not available in a rad-hard version. This part is designed for use with a 3.0v supply; here, we use it to monitor 3.1v. This supply voltage was chosen to be compatible with all of the following: the FPGA IO supply, the LMX2531 PLL (range 2.8 to 3.2), the crystal oscillator, the NAND flash (3.0 to 3.6), the LVDS transceivers (3.0 to 3.6).

The reset supervisor performs two functions. The critical function is to assert the POR_RESETb input of the Zynq low while the power supplies are coming up; if this is not done, the Zynq will not start up at all. A secondary function is that of watchdog supervisor: if the watchdog input is not toggled every second or so the watchdog output is asserted, which can perform a soft re-boot of the Zynq processor. I didn't implement the software for the watchdog function, but we may want this in a flight system.

There is a rad-hard version of this part, ISL706, for systems with a 3.3v supply, but this won't work with the 3.1v rail, as the "power-good" threshold can be as high as 3.15v. One option might be to use a comparator to assert POR_RESETb. Or use another LT3083 to make the 3.0v supply for the LMX2531, then use the ISL706. This might be the way to go if we want to use the watchdog timer.

The Latchup Current Limiter

As noted above, the LCL needs to be set to a level much higher than the operating current to permit the Zynq to start up. Maybe this is OK: the Zynq will survive an SEE-induced latchup which draws >1.3A. Or maybe we need to figure out how to set the LCL to a higher level during power-up and a lower one during operation.