# Saturn Project Notes

This document describes the FPGA for a new radio project using a Xilinx FPGA and an attached processing module. In the first instance this will be a Raspberry Pi 4 compute module but could be another embedded processor board. The interface to the FPGA will be PCI Express; the protocol code for data transfer to the “Thetis” PC will execute on the processor, not in the FPGA. The radio could be implemented on an oversize M.2 board, to fit onto several embedded processors.

This could be used in several ways:

|  |  |
| --- | --- |
| Diagram  Description automatically generated | The processor module is used simply to move data using protocol 1 – like the Red Pitaya’s processor. Low demand on the processor. Wired ethernet connection to PC running Thetis or other app. |
| Diagram  Description automatically generated | The processor module is used simply to move data using protocol 2. This will have higher demand because the data rate is higher. Wired ethernet connection to PC running Thetis or other app. |
| Diagram  Description automatically generated | The processor executes an SDR app such as Pihpsdr or linhpsdr. No PC required, and high quality display outputs are available. |
| Diagram  Description automatically generated | The processor execute Pihpsdr and has an attached 7” RPi touchscreen display. Possibly single ADC or 14 bit ADC version, with Apollo-like RF module. This could be a lower cost small radio, but there may be no market for it now. |

## FPGA Block Diagram

Diagram

Description automatically generated

Figure 1: Overall block Diagram

# Interfaces

## FPGA

|  |  |
| --- | --- |
| ADC | 2x LTC2208. Each with 5 bit attenuator. |
| DAC | The “normal” TX DAC. With 6 bit attenuator and 8 bit analogue PWM drive level. |
| Envelope DAC | Also want an “envelope” DAC for non linear TX amplifiers. Use an SPI driven DAC (but note not opto coupled). |
| Codec | TLV320AIC23B, as used on Hermes etc  I2S port driven by FPGA. the I2C port interfaced to the FPG but with data transactions initiated by the host processor. |
| Audio interfaces | Orion-like software settable connections to 3.5mm jack  Separate audio out for each of speaker and headphone, with separate gains  Balanced XLR mic input for “pro” audio people  With care, most of this could be on the rear panel I/O board with a ribbon cable to the FPGA board if there wasn’t enough space available. Fastest signal is MCLK (12.288MHz) |
| RF | SPI-like signals for Alex header. Follows the ANAN7000 standard, 16 bit TX, 32 bit RX. The analogue inputs from the RF board to ADC within the FPGA itself, if enough pins available. |
| Clock generation | 122.88MHz VCXO, with C/R filter on control voltage feed.  10MHz reference XTAL oscillator  10MHz external reference in, with auto select |
| PTT, keyer | 3 strobe inputs for PTT and dot/dash input. Buffered, and active low. |
| Config PROM | 256Mbit QSPI. Must be programmable through the FPGA with no jumper changes  Must be able to hold processor in reset until FPGA has configured. |
| JTAG | Standard Xilinx JTAG connector for debug |

## Processor (Raspberry Pi 4 Compute Module)

|  |  |
| --- | --- |
| PCIe | PCI express connection to FPGA. Gen2 x1. Need to look at the various signals associated with clock, reset etc. Provide additional PCI signals for X4; just not used with this module.  The design aim is for the connection to the radio to be entirely via PCIe; no other signals. This to make sure we can use other processor modules too. |
| USB | Need to be made available to user. |
| HDMI | Rear panel |
| Ethernet | Tracked to rear panel |
| Wi-Fi | Wi-Fi antenna connectors to connect to rear panel. |
| Power, reset | We need to investigate the power sequence / reset arrangements – if we can hold the processor in reset while the FPGA configures for perhaps 150ms, we can use a cheaper QSPI config prom. |

## RF Module Interfaces

### Alex header

|  |  |  |  |
| --- | --- | --- | --- |
| Pin | Function | Orion mk2: J15 | 7000 RF board: J7 |
| 1 | ANT\_TUNE | Buffered input to FPGA | (grounded) |
| 2 | +12V ALEX | +12V | Not connected |
| 3 | ALEX\_SPI\_SDO | SPI verilog | SPI registers |
| 4 | ALEX\_SPI\_SCK | SPI verilog | SPI registers |
| 5 | ALEX\_RX\_LOAD | SPI verilog | SPI registers |
| 6 | ALEX\_TX\_LOAD | SPI verilog | SPI registers |
| 7 | FWD\_POWER | Analogue in | Directional coupler FWD |
| 8 | PTT | Buffered input to FPGA | Not connected |
| 9 | REV\_POWER | Analogue in | Directional coupler REV |
| 10 | GND | GND | GND |

ANT\_TUNE and PTT appear to have no relevance; may be historical?

Note this is the only connection to the RF board. The RF board also has J22, for possible connection to an Arduino-like modules for LCD display.

# Power Supply, PCB Issues

This section covers issues related to the board & interfaces to the FPGA. Much of this is as per Orion mk2, with noted differences.

## Main interfaces- Changes from Orion mk2

This section notes deltas from the Orion mk2 design:

**DAC:** The DAC is driven by the FPGA which has differential LVDS outputs: the driver is not needed. This will be essentially as per Orion. (Shouldn’t there be 100R terminating resistors at the DAC between the +&- differential inputs?)

**ADCs:** The ADCs data to the FPGA is differential LVDS. There are 18 pairs per ADC (16 data + overflow + o/p clock). The shutdown pin SHDN is hardwired rather than FPGA driven: wire to 0v. Mode=2/3 VCC (2’s complement data, clock adjustment enabled). The LVDS pin connects to 3.3v to select the signal level. Other than that its schematic should be largely as per Orion. I assume there should be 100R terminating resistors at the FPGA between the +and- differential inputs.

**RF attenuators:** ADC attenuators “as is”. The DAC attenuator is an IDT F1912. Orion has pins to drive it in serial or parallel modes. Pluto will use “Direct Parallel” mode only to save pins. Wire VMODE (pin 13) to 0v; wire LE (pin 5) to 3.3v. The FPGA drives are 1.8V logic.

**7000DLE board interface:** The SPI data interface for RX and TX is from the FPGA. The analogue inputs (eg fwd, rev power) from the RF board go to an ADC internal to the FPGA.

**Codec & audio:** The CODEC I2C configuration interface needs to be driven by the FPGA. The discrete logic signals to control the audio path (eg tip/ring/bias etc) are on GPIO[4:0] pins. There is a new signal GPIO5, to select a differential XLR mic input amplifier. 2 spare signals GPIO6&7 if needed. If the CODEC were put on an I/O expander board (to save room on the main PCB) these could be driven by an I2C parallel register.

**Aux analogue inputs:** these use an internal ADC, so an SPI A-D converter is not needed.

## Floorplan

The current selected FPGA is Xilinx **XC7A200TFBG676-2**. It may be able to downsize to **XC7A100TFGG676-2** which is footprint compatible. I have allocated pins to the FPGA one the assumption that this approximate layout of main interfaces around the chip will be OK. IO pins are assigned to the FPGA banks to accommodate this. There are four sets of constraints:

1. The PCI Express data transceivers are fixed in bank 216 and can’t be changed;
2. The DAC LVDS output levels mean we can’t have CMOS 3.3V outputs in the same bank. (The DAC attenuator will be driven with lower voltage CMOS for that reason);
3. To enable moving down to the XC7A100T device, do not use banks 12 & 33;
4. Analogue inputs are in bank 35.

Chart, funnel chart

Description automatically generated

## Power Supplies

FPGA I/O levels are set “per bank”.

|  |  |  |
| --- | --- | --- |
| **Bank** | **Purpose** | **I/O Voltage** |
| Bank 0 | Configuration | 3.3V |
| Bank 13 | CODEC | 3.3V |
| Bank 14 | Config + ADC2 connections | 3.3V |
| Bank 15 | ADC1 connections | 3.3V |
| Bank 16 | DAC connections | 2.5V |
| Bank 34 | General logic | 3.3V |
| Bank 35 | General logic; analogue | 3.3V |
| Banks 12, 33 | Not used | 3.3V (I assume it still needs power) |
| GTP Quad 216 | GTP buffers (PCIe) | See below |

The analogue inputs (XADC) for PSU voltage/current, forward and reverse voltages connect to pins in bank 35.

Xilinx 7 series devices have power supply sequencing requirements. Supplies should come up in this order, & be shut down in reverse order:

|  |  |  |
| --- | --- | --- |
| 1 | VCCINT | 1V |
| 1 | VCCBRAM | 1V |
| 2 | VMGTAVCC[[1]](#footnote-1) | 1V (separate regulator, <10mV p-p noise) |
| 3 | VMGTAVTT[[2]](#footnote-2) | 1.2V (separate regulator, <10mV p-p noise) |
| 3 | VCCAUX | 1.8V |
| 3 | VCCADC | 1.8V |
| 4 | VCCO | Set per bank; 2.5V and 3.3V both needed (see above) |

The on-chip VCCADC supply is derived from VCCAUX: same voltage, but via a ferrite bead. See Xilinx UG480 figure 1-2 (p15). VREFP, VREFN, GNDADC are all at ground potential but with a ferrite bead to ground. 100nF and 470nF decoupling capacitors across VCCADC and GNDADC.

Diagram

Description automatically generated

## Decoupling

UG483 recommends a set of capacitors for each supply. These are package dependent. Assume that we will use XC7A200TFBG676.

|  |  |  |
| --- | --- | --- |
| **Pin** | **Voltage** | **Decoupling** |
| VCCINT | 1.0V | 1x 680uF; 1x100uF; 12x 4.7uF;17x 0.47uF |
| VCCBRAM | 1.0V |
| VCCAUX | 1.8V | 1x 47uF; 4x 4.7uF; 7x 0.47uF |
| VCCO\_0 | 3.3V | 1x 47uF |
| VCCO\_13 | 3.3V | 1x 47uF; 2x 4.7uF; 4x 0.47uF |
| VCCO\_14 | 3.3V | 1x 47uF; 2x 4.7uF; 4x 0.47uF |
| VCCO\_15 | 3.3V | 1x 47uF; 2x 4.7uF; 4x 0.47uF |
| VCCO\_16 | 2.5V | 1x 47uF; 2x 4.7uF; 4x 0.47uF |
| VCCO\_34 | 3.3V | 1x 47uF; 2x 4.7uF; 4x 0.47uF |
| VCCO\_35 | 3.3V | 1x 47uF; 2x 4.7uF; 4x 0.47uF |
| VCCO\_12 (unused | 3.3V | 1x 47uF; 2x 4.7uF; 4x 0.47uF |
| VCCO\_32 (unused) | 3.3V | 1x 47uF; 2x 4.7uF; 4x 0.47uF |
| MGTAVCC | 1.0V | 4.7uF ceramic; 2x 0.1uF |
| MGTAVTT | 1.2V | 4.7uF ceramic; 2x 0.1uF |
| MGTRREF\_216 | Reference input. Connect via 100R resistor to MGTAVTT | 4.7uF ceramic. |

Xilinx UG483 table 2-5 recommends suitable capacitors:

|  |  |
| --- | --- |
| 680uF Tantalum D package | T530X687M006ATE018 |
| 100uF tantalum or ceramic X7R 1210 | GRM32ER60J107ME20L |
| 47uF ceramic X7R 1210 | GRM32ER70J476ME20L |
| 4.7uF ceramic X7R 0805 | GRM21BR71A475KA73 |
| 0.47uF ceramic X7R 0603 | GRM188R70J474KA01 |

For placement of the MGTAVCC/MGTAVTT capacitors see Xilinx UG482 p230 (placement within the BGA pads is recommended)

## Pinouts

There is a separate spreadsheet “iopins.csv” created by the Xilinx tools with FPGA pinouts. This is a cross reference to that describing what the signals are for.

|  |  |  |  |
| --- | --- | --- | --- |
| **Pin(s)** | **Group/Interface** | **Dir’n** | **Comment** |
| BCLK | CODEC | Out | Bit clock to CODEC |
| LRCLK | CODEC | Out | L/R clock to Codec |
| MCLK | CODEC | Out | 12.288MHz master clock output to codec |
| i2stxd | CODEC | Out | Audio TX data to codec |
| i2srxd | CODEC | In | Audio RX data from codec |
| iic\_rtl\_0\_sda\_io | CODEC | Bidir-ectional | I2C signal to codec |
| iic\_rtl\_0\_scl\_io | CODEC | Bidir-ectional | I2C signal to codec |
| pcie\_reset\_n | PCI Express | In | Reset in from processor |
| pcie\_diff\_clock\_rtl\_clk\_n[0]  pcie\_diff\_clock\_rtl\_clk\_p[0] | PCI Express | In | PCIe reference clock. differential |
| pcie\_7x\_mgt\_rtl\_0\_rxn[3:0]  pcie\_7x\_mgt\_rtl\_0\_rxp[3:0] | PCI Express | In | PCIe RX data pairs |
| pcie\_7x\_mgt\_rtl\_0\_txn[3:0]  pcie\_7x\_mgt\_rtl\_0\_txp[3:0] | PCI Express | Out | PCIe TX data pairs |
| PCI\_LINK\_LED | PCI Express | Out | 0 (LED lit) when PCI link established. See 7.7 |
| PCIe\_CLK\_REQn | PCI Express | Out | Clock request output. Driven to 0V by configured FPGA. Needs pullup resistor. |
| MGTRREF\_216 | PCI Express | in | PCIe buffer reference. Needs 100R to MGTAVTT supply (see also section 3.4 & 3.9) |
| ADC1Ovr\_in\_N  ADC1Ovr\_in\_P | ADC | In | RF ADC1 overscale. Differential LVDS |
| ADC1\_In\_N[15:0]  ADC1\_In\_P[15:0] | ADC | In | RF ADC1 16 bit input data. Differential LVDS |
| ADC1\_CLKin\_N  ADC1\_CLKin\_P | ADC | In | RF ADC1 clock from ADC to FPGA  (not used but tracked; differential) |
| ADC2Ovr\_in\_N  ADC2Ovr\_in\_P | ADC | In | RF ADC2 overscale. Differential LVDS |
| ADC2\_In\_N[15:0]  ADC2\_In\_P[15:0] | ADC | In | RF ADC2 16 bit input data. Differential LVDS |
| ADC2\_CLKin\_N  ADC2\_CLKin\_P | ADC | In | RF ADC2 clock from ADC to FPGA  (not used but tracked; differential) |
| DAC\_Out\_N[15:0]  DAC\_Out\_P[15:0] | DAC | Out | RF DAC 16 bit output data. Differential LVDS |
| clock\_122\_in\_n  clock\_122\_in\_p | Sample Clock | In | 122.88MHz VCXO clock in; PECL with level shift |
| pll\_cr | Sample Clock | Out | PLL output to C-R circuit for VCXO control input |
| ref\_in\_10 | Sample Clock | In | 10MHz reference input from ext/oscillator select |
| TX\_DAC\_PWM | TX DAC related | Out | PWM o/p for DAC bias voltage (like DAC\_ALC) |
| Dac\_Atten[5:0]  Dac\_Atten\_LE  Dac\_Atten\_MODE  Dac\_Atten\_CLK  Dac\_Atten\_DATA | DAC attenuator | Out | TX DAC attenuator parallel data (see 3.1)  (1.8V logic signals)  (the serial wires are not used)  LE: drive to 1  MODE, CLK, DATA: drive to 0 |
| ADC1\_ATTEN\_CLK | ADC attenuator | Out | ADC1 atten control |
| ADC1\_ATTEN\_DAT | ADC attenuator | Out | ADC1 atten control |
| ADC1\_ATTEN\_LE | ADC attenuator | Out | ADC1 atten control |
| ADC2\_ATTEN\_CLK | ADC attenuator | Out | ADC2 atten control |
| ADC2\_ATTEN\_DAT | ADC attenuator | Out | ADC2 atten control |
| ADC2\_ATTEN\_LE | ADC attenuator | Out | ADC2 atten control |
| RF\_SPI\_CK | RF interface | Out | RF data control to Anan 7000 |
| RF\_SPI\_DATA | RF interface | Out | RF data control to Anan 7000 |
| RF\_SPI\_TX\_LOAD | RF interface | Out | RF data control to Anan 7000 |
| RF\_SPI\_RX\_LOAD | RF interface | Out | RF data control to Anan 7000 |
| Buf\_Alex\_Pin1 | RF interface | In | Unused Input |
| Buf\_Alex\_Pin8 | RF interface | In | Unused input |
| (see also configuration wiring diagram section 3.7) | FPGA Configuration |  |  |
| GPIO\_OUT[23:0] | Various | Out | Parallel data; see also section 7.5 |
| GPIO\_OUT[0] | Audio | Out | Mic bias enable. =1 to provide electret bias on 3.5mm jack |
| GPIO\_OUT[1] | Audio | Out | Input\_PTT\_Select  0=PTT on ring; 1=PTT on tip |
| GPIO\_OUT[2] | Audio | Out | Mic\_Signal\_Select  0=mic on ring, 1 = mic on tip |
| GPIO\_OUT[3] | Audio | Out | Mic\_Bias\_Select  0=bias on ring; 1= bias on tip |
| GPIO\_OUT[4] | Audio | Out | Spkr\_amp\_Mute |
| GPIO\_OUT[5] | Audio | Out | Balanced\_Mic\_Select  =1 to enable balanced mic input |
| GPIO\_OUT[7:6] | Audio | Out | Spare, uncommitted output |
| GPIO\_OUT[8] | RF ADC | Out | ADC1 RAND =1 to randomise data |
| GPIO\_OUT[9] | RF ADC | Out | ADC1 PGA =1 to enable ADC 3dB amplifier |
| GPIO\_OUT[10] | RF ADC | Out | ADC1 DITHER =1 to dither the clock |
| GPIO\_OUT[11] | RF ADC | Out | ADC2 RAND =1 to randomise data |
| GPIO\_OUT[12] | RF ADC | Out | ADC2 PGA =1 to enable ADC 3dB amplifier |
| GPIO\_OUT[13] | RF ADC | Out | ADC2 DITHER =1 to dither the clock |
| GPIO\_OUT[15:14] | RF | Out | Spare, uncommitted output |
| GPIO\_OUT[22:16] | General | Out | Open collector outputs (7 bits) |
| GPIO\_OUT[23] | General | Out | Spare, uncommitted output |
| DRIVER\_PA\_EN | TX Strobes | Out | Enables power to 0.5W amp |
| MOX\_strobe | TX Strobes | Out | 1=TX. See Orion FPGA\_PTT signal |
| TXRX\_RELAY | TX Strobes | Out | 0 if TX. Drive LED, lit for TX |
| CTRL\_TRSW | TX Strobes | Out | Drives relay by 0.5W amp |
| BUFF\_OUT | Strobe | Out | Unused. Wired to 0v |
| ATU\_TUNE | General CMOS | Out | =1 to initiate TUNE by external ATU |
| STATUS\_IN[9:0] |  | In | Parallel data; see section 7.6 |
| STATUS\_IN[0] | General CMOS | In | PTT In (3.5mm jack) |
| STATUS\_IN[1] | General CMOS | In | PTT In (rear panel) |
| STATUS\_IN[2] | General CMOS | In | Key in 1 (Dot) |
| STATUS\_IN[3] | General CMOS | In | Key in 2 (Dash) |
| STATUS\_IN[7:4] | General CMOS | In | User IO4,5,6,8 as drawn  IO5 used as a TX inhibit input  IO8 used as a CW input |
| STATUS\_IN[8] | General CMOS | In | 13.8v detect in. 1= power valid |
| STATUS\_IN[9] | General CMOS | In | ATU tune complete. 1= complete (needs pullup) |
| TX\_ENABLE | General CMOS | In | External input; if 0, TX is gated off. Needs pullup. |
| LEDOutputs[15:0] | Debug | Out | Drivers for status / debug LED. Active high, See section 7.7 |
| BLINK\_LED | Debug | Out | LED blinking at about 1Hz rate. See 7.7 |
| EMC\_CLK | Config | In | 122.88MHz CMOS clock in |
| PROM\_SPI\_ss\_io[0]  PROM\_SPI\_io3\_io  PROM\_SPI\_io2\_io  PROM\_SPI\_io1\_io  PROM\_SPI\_io0\_io  VCCBATT\_0  DONE\_0  CCLK\_0  INIT\_B\_0  M0\_0, M1\_0, M2\_0  PROGRAM\_B\_0  CFGBVS\_0  PUDC\_B | Config |  | Configuration signals. See diagram in section 3.7 |
| TCK\_0  TMS\_0  TDI\_0  TDO\_0 | Config |  | JTAG signals. See diagram in section 3.7 |
| ADC\_MOSI  ADC\_MISO  ADC\_CLK  nADC\_CS | Aux ADC; connects an SPI A-D converter |  | Pins added, and custom Verilog IP core. Not fully tested! |
| FPGA\_CM4\_EN |  | Out | Not currently used. Powers on/off the compute module 4. Drive to 1 for normal operation. |
| PCIe-T-SMBCLK  PCIe-T-SMBDAT |  | bidir | Not currently used: PCIe bus config signals |

A number of signals need pullup resistors (eg 4K7 to +3.3v):

STATUS\_IN[9]; TX\_ENABLE;

(not on the FPGA) RUN\_PG output to Raspberry pi4 CM

## Clocking

The clock arrangement should be as per Figure 2. The ADCs, DAC and FPGA should all get differential LVPECL clocks with trace lengths matched as far as possible to make the trace delays equal. The FPGA also requires a 122.88MHz clock to the EMCCLK pin (V22) used during configuration.



Figure 2: 122.88MHz Clock Distribution

Connect the FPGA LVPECL inputs with a similar resistor network to that on the ADCs. (The FPGA is set to LVDS levels – so we need PECL to LVDS translation).

ADC, DAC data timing at the FPGA can then be worked out (Figure 3):



Figure 3: Clock Timing For FPGA I/O

## FPGA Configuration

The FPGA loads its configuration from an external memory device at startup. By far the best arrangement will be to use a QSPI Flash device, as long as we can hold off the Raspberry pi starting up until FPGA configuration is complete. This will use Master SPI X4 Flash config mode (see Xilinx UG470).

Wire the JTAG and config PROM as per the Figure 4 & notes below:

Diagram, schematic

Description automatically generated

Figure 4: Schematic for Configuration Prom

|  |  |  |
| --- | --- | --- |
| **Pin** | **Package Pin** | **Comment** |
| D[00] | P22 | in-system programming of QSPI Prom. Signals PROM\_SPI\_MOSI, PROM\_SPI\_MISO, PROM\_SPI\_SSn[0] in the FPGA pin list connect to these 3 pins. |
| DIN/D[01] | R22 |
| FCS\_B | T19 |
| D[02] | P21 |  |
| D[03] | R21 |  |
| CCLK | L12 | Does not appear in the FPGA pin list |
| INIT\_B | U12 |  |
| DONE | G11 | Drives Raspberry Pi RUN\_PG via an open collector buffer and jumper |
| TDO | U13 | Use a 2mm pitch 2x7 pin header for Xilinx download cable. Xilinx suggested parts (DS593 page 15):  Molex 87832-1420 (SMT) or 87831-1420 (leaded)  FCI 98424-G52-14 (SMT) or 98414-G06-16 (leaded) |
| TDI | R13 |
| TMS | T13 |
| TCK | V12 |
| PROGRAM\_B | N12 |  |
| M0 | U11 |  |
| M1 | U10 |  |
| M2 | U9 |  |
| EMCCLK | V22 | 3.3V CMOS clock for configuration only |
| PUDC\_B | U22 |  |
| CFGBVS | U8 | Selects the correct configuration voltage. |
| VCCO\_0 | F12, T12 | 3.3V power rail |

The S25FL256 devices are ready for read 300us after power applied. This is significantly faster than the FPGA TPOR (10-35ms) so the Flash memory will be ready when the FPGA begins.

The XC7A75T & XC7A100T both require approx. 30Mbit configuration data. The XC7A200T requires 75Mbit. Xilinx has moved away from specific flash devices – instead supporting commodity devices. The de-facto config prom is a QSPI device costing perhaps £2 and consuming only 8 pins. These read on quad output, fast read mode; not DDR. XAPP586 describes using a QSPI serial prom in master SPI mode with 7 series devices.

UG908 lists flash PROMs specifically supported by Xilinx. Cypress S25FL128L/256FL, ISSI Is25LP128, IS25WP128 all suitable & available from Farnell. These can clock at 133MHz but including setup time into FPGA, 80MHz is more realistic. Using 61.44MHz clock XC7A75T would configure in approx. 125ms in quad mode.

PCI Express has a commonly quoted “must be configured in 100ms” requirement. Actually it’s nearer 70ms:

* A PC PSU requirement is 100ms from power good until the power supply releases reset (we might have some flexibility here, as we aren’t using a PC power supply);
* PCIe says be ready for configuration 20ms after that;
* But Xilinx on-chip reset may take 50ms to release after it senses power OK.

QSPI unlikely to meet that requirement on its own BUT if we can hold off the CPU reset for a further 200ms say (Can we hold RUN\_PG low until ready – needs to be an open drain signal?), a cheap QSPI device would be fine. The on-chip RC oscillators have huge variation in clock rate caused by temp & process variation. We do have a good quality 122.88MHz VCXO available, which can be divided.

UG953 & XAPP1020 describes how to access the reserved configuration pin CCLK to be able to program the PROM: need to use a STARTUPE2 primitive.

There is an app note XAPP518 that covers in-system programming the flash device through a PCI express endpoint. There’s code for a host application to read the BIT file; and code for the FPGA to provide the data path. There is a mechanism to provide an “emergency” configuration in case the write process is interrupted and the code corrupted.

There are settings to be made in Vivado to specify config details into the bitstream. This will cover selecting the x4 data width and switching to the EMCCLK pin with a correct division ratio. See XAPP586.

## Reset Arrangements

We will need to check what reset arrangements are appropriate. The PCI express reset might not be appropriate, if this resets the FPGA causing it to reconfigure. Check what other boards do. Ideally we want the FPGA to be released from reset first, allowing it to configure THEN releasing the processor from reset (which is what the DONE signal above would achieve). To be on the safe side, the PCI express reset PCIe\_nRST and the reset to the Raspberry Pi computer module RUN\_PG should both be jumpered and have pullup resistors.

## Processor PCI Express interface

Ideally we should allow a 4 lane PCI express interface to allow for future expansion. Initially we will use 1 lane Gen2 (Gen2x1); Allow eventually for up to Gen2x4. The PCI Express bus signals are:

|  |  |
| --- | --- |
| PCIe\_nRST | Reset signal, driven by processor |
| PCIECLKREQN | Clock request. Permanently low output from FPGA, once configured. |
| PCIe\_CLK\_P,  PCIe\_CLK\_N | Differential clock; 100MHz as required by PCI Express spec; HCSL levels. Clocks the IP core. AC coupled at the RPi source. |
| PCIe\_TX\_P,  PCIe\_TX\_N | Differential RPi data out; to FPGA data in. AC coupled at the RPi source.  Lane 0 only used for RPi4. |
| PCIe\_RX\_P,  PCIe\_RX\_N | Differential RPi data in; from FPGA data out. AC coupled near the FPGA; use 0.1uF X7R. Lane 0 only used for RPi4. |
| MGTRREF\_216 | Needs 100R to MGTAVTT supply (see also section 3.4) |

Note that the ordering of the high speed GTPE2 transceivers is set in the XDMA core (actually the hard PCIe block). The pinouts can’t be reassigned.

See NVidia OEM design guide for advice on tracking: it is probably still relevant. Raspberry pi recommendation is for 90Ω Traces, and with lengths within a pair matched within 0.1mm

Diagram, schematic

Description automatically generated

Figure 5: PCI Express Signals

PCIe\_CLK\_nREQ: PCI express “clock request” to processor. 3.3V level. Needs a pullup resistor; driven to 0 by an FPGA output.

PCIe\_nRST: active low PCI express reset from processor. Treat as a CMOS 3.3V level input. This should reset the PCI express IP core in the FPGA.

The PCIe core has “preferred” GTP transceivers:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **PCIe Lane** | **GTP transceiver** | **Pin names**  **XC7A100T-FGG484** | **Pin Names**  **XC7A100T-FGG676** | **Pin Names**  **XC7A200T-FBG676** |
| Lane 0 | X0Y7 | D7 MGTPTXP3\_216  C7 MGTPTXN3\_216  D9 MGTPRXP3\_216  C9 MGTPRXN3\_216 | D10 MGTPTXP3\_216  C10 MGTPTXN3\_216  D12 MGTPRXP3\_216  C12 MGTPRXN3\_216 | D10 MGTPTXP3\_216  C10 MGTPTXN3\_216  D12 MGTPRXP3\_216  C12 MGTPRXN3\_216 |
| Lane 1 | X0Y6 | B6 MGTPTXP2\_216  A6 MGTPTXN2\_216  B10 MGTPRXP2\_216  A10 MGTPRXN2\_216 | B9 MGTPTXP2\_216  A9 MGTPTXN2\_216  B13 MGTPRXP2\_216  A13 MGTPRXN2\_216 | B9 MGTPTXP2\_216  A9 MGTPTXN2\_216  B13 MGTPRXP2\_216  A13 MGTPRXN2\_216 |
| Lane 2 | X0Y5 | D5 MGTPTXP1\_216  C5 MGTPTXN1\_216  D11 MGTPRXP1\_216  C11 MGTPRXN1\_216 | D8 MGTPTXP1\_216  C8 MGTPTXN1\_216  D14 MGTPRXP1\_216  C14 MGTPRXN1\_216 | D8 MGTPTXP1\_216  C8 MGTPTXN1\_216  D14 MGTPRXP1\_216  C14 MGTPRXN1\_216 |
| Lane 3 | X0Y4 | B4 MGTPTXP0\_216  A4 MGTPTXN0\_216  B8 MGTPRXP0\_216  A8 MGTPRXN0\_216 | B7 MGTPTXP0\_216  A7 MGTPTXN0\_216  B11 MGTPRXP0\_216  A11 MGTPRXN0\_216 | B7 MGTPTXP0\_216  A7 MGTPTXN0\_216  B11 MGTPRXP0\_216  A11 MGTPRXN0\_216 |
| (the preferred GTP transceivers are the same for each of: XC7A100TFGG484, XC7A100TFGG676, XC7A200TFBG676; but the pins numbers change between packages) | | | | |

# FPGA Internal Design Notes - Conventional Time Domain Processing

The board will be used in one of 3 ways:

1. With an external application, data sent over ethernet using protocol 1 by Raspberry pi. The data interface should follow protocol 1 (ie 5-7 interleaved down converters)
2. With an external application, data sent over ethernet using protocol 2 by Raspberry pi. This is the most stressing case and the data format at the FPGA interface should try to match protocol 2 data structures to minimise processor load.
3. With the Raspberry pi running an SDR application (eg pihpsdr). The data interface doesn’t need to follow any published protocol, but protocol 2 is easiest to unpack

We know that a lesser processor can handle the data transfers for protocol 1; so managing the various control etc data registers won’t be a problem. It is only the main sample flows that need to be designed carefully.

## FPGA Requirements

* Use PCI express to communicate with host computer
* TX function
  + Accept I/Q baseband TX samples
    - 48KHz, 16 bit I/Q (protocol 1)
    - 192KHz 24 bit I/Q (protocol 2)
  + Upconvert to required TX frequency
  + Present samples to DAC
  + Be able to gate off the DAC samples when TX strobe not asserted
  + Be able to TX a test source waveform
  + For CW modes – ramp amplitude up, down as key pressed/released
* Downconvert RX
  + Have at least 5 separate RX DDC
  + Each DDC select samples from ADC1, ADC2, TX samples, RX test source
  + Separate selection for each of RX and TX (current design does this, but it goes beyond what protocol 2 requires)
  + Downconvert to baseband I/Q
  + Present samples to PC for each DDC streams
  + Data format will be protocol dependent. Look at FIFO sizes needed.
  + Selectable sample rate 48/96/192/384/768/1536KHz
  + Accept “overscale” input from ADC & latch
  + Present latched “overscale” to processor
  + Provide a way to allow the CPU to clear the overrange indication
* Clock Management
  + Accept 10MHz ref signal from local xtal or external input
  + Control 122.88MHz VCXO
  + Take clock inputs from VCXO, ADC
  + Generate 122.88MHz DAC clock
  + Derive CODEC master clock (12.288MHz)
* CODEC interface
  + CODEC setup will be via I2C; connected to the FPGA but software on the host will write the registers
  + Accept 48 KHz mic scalar samples from CODEC, send via FIFO to host computer
  + Accept 48KHz L/R speaker samples, send via FIFO to CODEC
  + When in CW mode, add to the TX path a CW sidetone
  + CW sidetone from freq programmable DDS, amplitude ramps up and down to avoid keyclick.
  + Overall sidetone amplitude set according to user preference
* RF interface
  + Provide interface signals for ANAN7000 series RF
  + Drive SPI-like interface for RX (set entirely by processor)
  + Drive SPI-like interface for TX (set by processor, updated by TX strobe)
  + Send control signals to RF attenuators
* Configuration
  + Be able to read back various strobes
  + Be able to assert TX strobe, FIFO resets, control lines for TX & RX
  + Be able to rewrite FPGA configuration PROM without jumper change
  + Be able to reset all FIFOs
  + Be able to read back a version number in some form
* I/O Signals
  + Accept PTT, Keyer signals and send to host PC
  + Accept analogue inputs: either attached ADC or Xilinx ADC
  + Drive I2C for codec etc

## Sampling Architecture

### RX

Uses conventional 122.88MHz clock; 16 bit ADC/DAC; designed for 2 ADC. The required Output Rates are:



I plan to adopt the concept of Phil Harman’s approach with CIC filter followed by decimate-by-8 FIR. That means the CIC is used well within its passband and is almost flat (0.4dB droop predicted at +- Fs/2).



Consequently the FIR has the same shape and the same coefficients can be used for each sample rate setting.



Figure 6: CIC Decimate by 40, 6 stages (FIR width shown in red)

The CIC is decimate by 40 for final Fs=384KHz. Passband droop at +/-192KHz is ~ 0.4dB. The FIR is needed to accelerate the cutoff at the Fs/2 point; the number of taps driven by the sharpness desired.

The FIR aliases at its input sample rate Fs. For the diagram above for final Fs=384KHz, FIR Fs = 8\*384 = 3072KHz. So the filter aliases (at 3072KHz , 6144KHz etc) map onto the nulls in the CIC spectrum. The filter width is narrow; the CIC provides ~140dB rejection at the FIR alias points.

6 stage CIC, decimating by a variable rate

### TX

TX has fewer choices. 16 bit DAC; 16 bit I/16 bit Q input samples @Fs=48KHz (protocol1) or 24 bit I/24 bit Q input samples @192KHz (protocol2) currently I’ve only implemented 16 bits

The DDS does not need to be used in “unit circle” mode. Amplitude variation negligible after filters have settled.



From the diagram (Figure 7) the operation of the decimating filter is clear. The bandwidth is defined at the start; decimation simply reduces the sample rate to 1/8 of the original. The spectrum display will be +/- Fs/2. A signal just above Fs/2 will alias to being just inside the passband at just above -Fs/2. So we need the filter to pass the required signals inside +/-FS/2 and reject others to the stop band required. Suggest a stopband depth of 100 to 120dB is appropriate.

For final Fs=48KHz we could consider having the filter select 40KHz of “useful” spectrum. That would make the cutoff +/-20KHz.



Figure 7: Decimate by 8 Filter

## Receiver Implementation

The receiver currently contains 5 DDCs; each channel can accept data from ADC1, ADC2, TX samples or a test source. I/Q samples from all 5 channels are combined into a single data FIFO or may be to separate FIFOs. (This needs to be finished off!) The receiver is clocked at 122.88MHz and each DDC has an individually selectable sample rate.

### DDC Architecture

The DDC is the receiver building block; it outputs downconverted and decimated / filtered I/Q samples having processed an ADC input stream. The Vivado block design flowgraph is shown in Figure 8 and Figure 9. The first is unfortunately hard to read! The DSP is implemented entirely using Xilinx IP blocks provided as part of the Vivado package. Each IP core is customised according to user entered parameters. The DSP cores are marked in red in the first figure; the rest are infrastructure.



Figure 8: DDC Flowgraph in Vivado

Diagram

Description automatically generated

Figure 9 Single Channel DDC

The processing is as follows:

* Input samples are chosen from 4 sources according to the channel select bits. Sources available are:
  + ADC1
  + ADC2
  + TX samples that go to the DAC
  + A “test sources” DDS
* The downconversion frequency is generated using a quadrature DDS; its NCO sets the frequency, and sin/cos lookup tables generate I/Q. The DDS free-runs at 122.88MHz. The DDC parameters are:
  + SFDR 95dB (implies 16 bit I/Q output)
  + Frequency resolution 0.05Hz (implies 32 bit phase accumulator)
* The selected sample stream connects to one port of a complex multiplier. The data sources provides the I samples; the Q samples are zeroed. The other port connects to the 16 bit I/Q DDS. The complex multiplier parameters are:
  + 16+16 bit I/Q inputs
  + 16+16 bit I/Q outputs
  + Set to truncate to set output data width (random rounding is another option)
* The output of the complex multiplier is an I/Q stream at zero centre frequency. It now needs to be filtered and decimated to the final bandwidth and matching sample rate. A dual stage filtering scheme is used with CIC decimating to 8x the final sample rate, then a further decimate-by-8 FIR to set the final output sample rate.
* The complex sample stream is split into two separate scalar streams
* The I/Q streams each have their own CIC filter, with decimation set in binary steps from 10 to 320 depending on required final sample rate. The CIC parameters of the core are:
  + 6 CIC stages
  + Differential delay = 1
  + Output data width = 18 bits
* The two scalar filtered streams are recombined into a single complex stream. An FIR filter IP processes both the I and the Q streams, re-using its multiplier and coefficient storage resources. The coefficients were generated using a web filter design site, and converted to .coe format using an excel spreadsheet. The core takes floating point coefficients, and normalises them. The FIR parameters are as follows:
  + 512 taps
  + Coefficient file: 512tap\_TX\_filter\_tfilter.coe (yes I know it says TX!)
  + Decimate by 8
  + Coefficient width 22 bits, fractional bits 24
  + Output width 28 bits
* The output data is limited to 24 bits by taking the 23 LSBs (full amplitude is never reached).
  + The bottom 23 bits are selected at output bits (23:1); output bit 0 = 0
  + (This gives near full amplitude output data from full amplitude input sinewave)
* The single stream is expanded back to an I/Q stream giving 24+24 bit I/Q data at the required final sample rate.

In protocol 2 each receiver can have a different sample rate, so the same FIR can’t be shared across all receivers. Each receiver slice has one FIR shared by I and Q. The max input sample rate is 12.288MHz (CIC decimating by 10, FIR decimate by 8 to give 1536KHz I/Q sample rate). The filter implements 2 channels. The final AXI stream width change gives parallel I/Q data output at the required sample rate.

Each DDC FIR uses 14 DSP48 slices and 7 Block RAMs. The CIC filters have been set to not use DSP48 slices and use FPGA logic instead; it seems to make little overall impact: it seems to fit combinatorial logic well. If the filter was not shared between I and Q we would need 2 FIR filters, each 8 DSP48 slices and 4 block RAMs. There is a marginal saving by using a shared filter.

The filter has been time domain simulated to check the word sizes. (I also found at the same time that the DDS needs a reset strobe, or its output is forever XXXX because it has accumulated XXXX when starting up). We get a useful 23 bits out of the FIR. To provide 24 bit output, use a 32 to 64 bit aix stream data width converter then a subset converter with the tdata remap string set to: tdata[54:32],1'b0,tdata[22:0],1'b0

DDC registers:

The channel config data bits select the required input sample stream.

|  |  |  |
| --- | --- | --- |
| **Input Bits** | **Function** | **Meaning** |
| Chan\_Config(1:0) | Select the input when in both RX and TX modes | 00: ADC1  01: ADC2  10: Test source  11: TX samples |

### Combined 5ch RX

(to be updated for protocol 2!)



Figure 10: Multi Channel Receiver

### Receiver Registers

|  |  |  |
| --- | --- | --- |
| **RX Config Register** | | |
| **Input Bits** | **Function** | **Meaning** |
| RXConfig(1:0) | Channel configuration for RX1 | See table for RX channel |
| RXConfig(3:2) | Channel configuration for RX2 | See table for RX channel |
| RXConfig(5:4) | Channel configuration for RX3 | See table for RX channel |
| RXConfig(7:6) | Channel configuration for RX4 | See table for RX channel |
| RXConfig(9:8) | Channel configuration for RX5 | See table for RX channel |
| RXConfig(12:10) | RX1 sample rate register | Sets CIC decimation  0: 48KHz; 1: 96KHz; 2: 192KHz  3: 384 KHz; 4: 768 KHz; 5,6,7: 1536 KHz |
| RXConfig(15:13) | RX2 sample rate register | Sets CIC decimation  0: 48KHz; 1: 96KHz; 2: 192KHz  3: 384 KHz; 4: 768 KHz; 5,6,7: 1536 KHz |
| RXConfig(18:16) | RX3 sample rate register | Sets CIC decimation  0: 48KHz; 1: 96KHz; 2: 192KHz  3: 384 KHz; 4: 768 KHz; 5,6,7: 1536 KHz |
| RXConfig(21:19) | RX4 sample rate register | Sets CIC decimation  0: 48KHz; 1: 96KHz; 2: 192KHz  3: 384 KHz; 4: 768 KHz; 5,6,7: 1536 KHz |
| RXConfig(24:22) | RX5 sample rate register | Sets CIC decimation  0: 48KHz; 1: 96KHz; 2: 192KHz  3: 384 KHz; 4: 768 KHz; 5,6,7: 1536 KHz |
| RXConfig(30) | ADC1 clear overflow | =1 to clear overflow latch |
| RXConfig(31) | ADC2 clear overflow | =1 to clear overflow latch |
| RX1Tune(31:0) | RX1 DDS Tune | 32 bit phase word |
| RX2Tune(31:0) | RX2 DDS Tune | 32 bit phase word |
| RX3Tune(31:0) | RX3 DDS Tune | 32 bit phase word |
| RX4Tune(31:0) | RX4 DDS Tune | 32 bit phase word |
| RX5Tune(31:0) | RX4 DDS Tune (also test source) | 32 bit phase word |

The ADC has a single overflow bit, valid in the cycle where overflow occurred. Needs to be latched, and cleared on processor request.

ADC randomise inverts bits 15:1 if bit0 is 1.

## Transmitter Implementation

The transmitter is clocked at the full output sample rate (122.88MHz). It ultimately provides sample data to the TX DAC (MAX5891) with offset binary data format. That is simply converted from 2’s complement by inverting the MSB.

### DSP Architecture

The TX uses the same approach; interpolate by 8 FIR then a CIC interpolator. For CW TX, the DDS is adjusted to offset by the sidetone frequency. CW keying simple scales its amplitude through the I/Q sample path. We don’t need a TX test source – just need to be able to turn on the DDS output at selectable amplitude.

For protocol 1 the TX sample rate is 48KHz, 16+16 bit I/Q. For Protocol 2 it is 192KHz, 24 + 24 bit I/Q. I have only implemented 16+16 bits at the moment.

The TX flowgraph is shown in Figure 11 and Figure 12. This includes a partly implemented path for TX envelope generation, for EER; but this is incomplete. (I know for example it needs a different I/Q feed with a delay). The TX again uses Xilinx IP cores, marked in red.



Figure 11: Transmitter Vivado Flow Graph



Figure 12: Transmitter

The TX signal path is as follows:

* I/Q modulation samples are selected from 1 of 4 sources:
  + The TX samples from the DSP application (eg Thetis)
  + A test DDS source;
  + A CW keyer;
  + A fixed amplitude, 0Hz sample.
* The I/Q samples are multiplexed into a single scalar data stream.
* An interpolating FIR filter bandwidth limits the samples and increases the same rate. Like the RX, the FIR filter has designed using a web filter designer, then the Xilinx coefficient file is generate using an excel spreadsheet. The FIR filter parameters are:
  + Input sample width 16 its
  + 512 filter taps
  + Coefficient file: 512TapLPF\_corner\_20KHz\_tfilter.coe
  + Interpolate by 8
  + Coefficient width 22 bits, fractional bits 24
  + Output width 20 bits
* Filtered samples are converted back to an I/Q stream
* The stream is split into separate I and Q samples.
* Each stream is filtered by identical interpolating CIC filters. The CIC parameters are as follows:
  + 6 stages
  + Differential delay = 1
  + Interpolate by either 80 (protocol2) or 320 (protocol1)
  + Input data width 20 bits
  + Output data width 23 bits
* The filtered samples, now at the final sample rate of 122.88MHz, are converted back into a single I/Q stream and connected to one port of a complex multiplier.
* A quadrature DDS generated samples of the local oscillator for upconversion. The DDS parameters are:
  + SFDR 95dB (implies 16 bit I/Q output)
  + Frequency resolution 0.05Hz (implies 32 bit phase accumulator)
* A complex multiplier multiples the complex modulating samples by the complex DDS samples. The multiplier parameters are:
  + Channel A (modulation) width 23 bits
  + Channel B(DDS) width 16 bits
  + Output width 20 bits
* The In-phase output is selected and scaled in amplitude by a processor-defined 18 bit word. The top 16 bits are taken and used to drive the output DAC. The samples can be gated to 0 when TX is not in progress. The DAC samples are also passed back to the receiver to be downconverted for Puresignal processing.
* Although not fully implemented, a cordic IP core extracts the TX signal magnitude. This would then be used to generate the envelop output signal using a PWM DAC. Alternatively an SPI DAC could be used, but this is not included in the current design.

### Transmitter Registers

|  |  |  |
| --- | --- | --- |
| **TX Config Register registers** | | |
| **Input Bits** | **Function** | **Meaning** |
| TXLOTune[31:0] | TX DDS frequency | 32 bit phase word |
| TXTestFreq[31:0] | Test source tune word |  |
| TXConfig[1:0] | Select the TX data source | 00: TX I/Q Data  01: Fixed amplitude 0Hz  10: Test DDS source  11: CW keyer |
| TXConfig[2] | Output sample gating | 0: TX/RX controlled  1: always on |
| TXConfig[3] | Protocol | 0: protocol 1  1: protocol 2  (selects interpolation rate) |
| TXAmpl[21:4] | Output amplitude | 18 bit ampl word, applied at DAC |
| TXConfig[4] | TX FIFO reset | =1 to reset |
| TXConfig[5] | RX FIFO reset | =1 to reset |
| TXConfig[6] | CODEC FIFO reset | =1 to reset |

The TX LO DDS is 29 bits with a resolution~0.4Hz. The I/Q test source DDS is 29 bits with a resolution ~0.4Hz

Not sure about the FIFO resets!

### EER

The EER function uses the TX signal envelope, a few times faster than TX I/Q sample rate (Orion is 5Fs). Currently I calculate the envelope at the full DAC output rate which will need to be decimated to a suitable DAC speed.

Suggest using an SPI DAC with 12 bit resolution. There is available Verilog code for an axi stream to SPI IP core. Use the Verilog code to decimate the o/p sample rate to an acceptable rate (eg 384kS/s, compatible with protocol 1 and 2 rates). A suitable DAC is MCP4821.

Note that there is no code in Orion to drive this signal; we may be able to remove it completely.

## Codec & Audio Interface

### Architecture

This interfaces to a TLV320AIC23B codec (same as Hermes). It uses I2S Slave mode, with timing strobes derived by the FPGA. An I2C (2 wire) interface connected to the processor is used for configuration.

Audio sample rate = 48KHz for both protocols.



Figure 13: CODEC Interface

### Codec Clocks

The Codec has several clocks that the FPGA needs to generate. All are synchronous to the 122.88MHz FPGA and sample clock. The audio sample rate is 48KHz.

The Codec has a master clock of 12.288MHz; sample clock divided by 10. This is generated by a Xilinx clock generator.

There is a data clock BCLK, generated by the I2S interface logic.



### Codec Registers

|  |  |  |
| --- | --- | --- |
| **Codec Config Register registers** | | |
| **Input Bits** | **Function** | **Meaning** |
| CodecConfig(31:16) | Sidetone volume | 16 bit ampl word, unsigned |
| CodecConfig(15:0) | Sidetone frequency | 16 bit phase word (note 48KHz effective Fs) |

The Codec interface and the TX need to maintain constant latency regardless of whether CW or other modes are used. Sidetone is added to the speaker path, not replacing it (this avoids clicks through gating off an active audio signal).

The circuit including DDS is clocked at 12.288MHz rate. But the DDS is “throttled” by the data rate that the I2S interface will accept (ultimately 48KHz word rate) and the TREADY signal as part of the AXI stream interface sets the effective clock rate.

(CW PTT in can be ignored completely in the processor)

### Codec I2C Register Settings

Taken from the Hermes “hermes\_TLV320\_SPI.v” code: these settings will need to be made by the processor at power up. Also some settings at runtime.

The CODEC uses two bytes for a register write: a 7 bit register address and 9 bit register data.

|  |  |  |
| --- | --- | --- |
| **Register (hex)** | **Value (9 bits, hex)** | **Meaning** |
| 0F | 000 | Reset device |
| 09 | 01 | Digital interface activation: set to ACTIVE |
| 04 | 10: line  14: mic no boost  15 mic, boost | Analogue audio path control  Line: mic not muted; line input; bypass disabled; sidetone disabled  Mic: mic not muted; mic input; bypass disabled; sidetone disabled  (set bit 0 for 20dB boost) |
| 06 | 00 | Power down control.  All elements powered on |
| 07 | 02 | Digital interface format.  Slave; no swap; right when LRC high; 16 bit; I2S format |
| 08 | 00 | Sample rate control  No clock divide; sample rate ctrl=0; normal mode, oversample 256Fs (suitable for MCLK=12.288MHz, 48KHz ADC & DAC) |
| 05 | 00 | Digital audio path control  DAC soft mute disabled; de-emphasis disabled; ADC high pass filter enabled |
| 00 | Line in gain??  0000nnnnn | Left line input volume  No mute; no simultaneous update; gain=nnnnn |

### Codec Hardware Interface

|  |  |  |
| --- | --- | --- |
| **CODEC Pin** | **Connection** | **Function** |
| MODE | Hardwired to 0 | Selects I2C |
| CS~ | Hardwired to 0 | Selects address = 0x1A |
| SCLK | FPGA I2C\_SCK | I2C clock; 400KHz |
| SDIN | FPGA I2C\_SDA | I2C data |
| MCLK | FPGA MCLK | 12.288MHz clock |
| BCLK | FPGA BCLK | I2S bit clock |
| LRCIN  LRCOUT | FPGA LRCLK | Left/right select. Both driven by the same FPGA signal. |
| DIN | FPGA I2STXD | I2S serial speaker audio data to CODEC |
| DOUT | FPGA I2SRXD | I2S serial microphone data from CODEC |
|  |  |  |

# Clock Generation

The TX and RX sample path is clocked by a 122.88MHz VCXO. That is phase locked to a 10MHz reference.

## FPGA Clocks

The FPGA uses two clocks:

122.88MHz ADC/DAC sample clock, for all RX/TX paths

Derived 12.288MHZ clock for audio codec, debounce, SPI data shifting etc. Used as the CODEC MCLK source.

The 122.88MHz sample clock may have three different phases in the FPGA:

* The “main” clock input
* The ADCs have a “clock output” connected to the FPGA. Those can be used to register samples into the FPGA; those signals will be an additional “clock capable” input to the FPGA.
* The DAC may need its data driving from a different clock phase to meet the output timing.
* The timings for all of these need to be worked out!

## PLL

The phase lock is:

1. Divide 122.88MHz clock by 3072
2. Divide 10MHz clock by 250
3. Exor the two signals
4. That output goes to the loop filter on the VCXO control voltage.

The Orion board has an auto detector for external reference, and automatically selects it if detected.



Figure 14: PLL For 122.88MHz VCXO

The VCXO control output needs the same C/R filter that Hermes etc have.

## AXI Bus

The PCI Express interface receives a 100MHz clock. The AXI output from the PCIe core has a 125MHz clock rate.

The AXI Buses after the AXI interconnect operate at two speeds:

1. Those associated with the radio hardware operate at 122.88MHz;
2. Those providing processor peripherals operate at 125MHz.

# CW Keyer

Keyer Verilog code designed using code from profile.v and Pavel Demin’s code. The ramp rate is throttled from the axi stream tready. Keyer also generates a PTT signal.

Two keyers needed: one for audio sidetone and one for the TX ramp. Ramp time is 5ms (960 samples @protocol 2 192KHz, 240 samples @ audio/protocol 1 48KHz)

|  |  |  |
| --- | --- | --- |
| **CW Keyer Register** | | |
| **Input Bits** | **Function** | **Meaning** |
| CW\_Keyer[7:0] | CW PTT Delay | 0-255 ms; units ms |
| CW\_Keyer[17:8] | CW Hang time | 0-1023 ms; units ms |
| CW\_Keyer[18] | Enable | 0=off; 1=enabled |

# RF System Control & GPIO

This function is clocked at 12.288MHz (Fs/10). To be included into the FPGA block:

* 2x5 bit atten control output
* 6 bit atten control output
* DAC drive level PWM output
* Aux DAC output (EER)
* RX/TX SPI control

## RF SPI Interfaces

The radio uses the ANAN7000DLE RF hardware, and its SPI control interface. Two words are used – 16 bits TX, and 32 bit RX. The data is transferred to the radio whenever a change in data bits is detected.

Most significant bit shifted first

U3/U5: TX Settings: 16 bit SR. Serial data = ALEX\_SPI\_SDO; Serial clock = ALEX\_SPI\_SCK; LOAD clock = ALEX\_TX\_LOAD

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **TX\_SPI[15:0]** | | | | | |
| **Bit** | **Function** | **Notes** | **Bit** | **Function** | **Notes** |
| 0 | N/A |  | 8 | ANT1 |  |
| 1 | N/A |  | 9 | ANT2 |  |
| 2 | TXRX\_STATUS | unsure | 10 | ANT3 |  |
| 3 | LED D9 |  | 11 | TXRX\_RELAY | Operates T/R relay. 1=TX |
| 4 | BPF3 | 20-30m | 12 | LED-D7 |  |
| 5 | BPF2 | 40-60m | 13 | BYPASS | 6m |
| 6 | BPF1 | 80m | 14 | BPF5 | 10-10m |
| 7 | BPF0 | 160m | 15 | BPF4 | 15-17m |

(Note bit 11 isn’t a CPU register and needs to be replaced by TX/RX signal)

U6/U10/U7/U13: RX Settings: 32 bit SR. Serial data = ALEX\_SPI\_SDO; Serial clock = ALEX\_SPI\_SCK; LOAD clock = ALEX\_RX\_LOAD

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **RX\_SPI[31:0]** | | | | | |
| **Bit** | **Function** | **Notes** | **Bit** | **Function** | **Notes** |
| 0 | YELLOWLED |  | 16 | YELLOWLED 2 |  |
| 1 | 13HPF | 10-22MHz BPF | 17 | 13HPF 2 | 10-22MHz BPF |
| 2 | 20HPF | 22-35MHz BPF | 18 | 20HPF 2 | 22-35MHz BPF |
| 3 | 6MLNA | 50MHz BPF&LNA | 19 | 6MLNA 2 | 50MHz BPF & LNA |
| 4 | 9.5HPF | 6-10MHz BPF | 20 | 9.5HPF 2 | 6-10MHz BPF |
| 5 | 6.5HPF | 2.5-6MHz BPF | 21 | 6.5HPF 2 | 2.5-6MHz BPF |
| 6 | 1.5HPF | 1-2.5MHz BPF | 22 | 1.5HPF 2 | 1-2.5MHz BPF |
| 7 | N/A |  | 23 | N/A |  |
| 8 | XVTR RELAY | Transverter in | 24 | RX2\_GROUND | When 1, RX2 i/p disconnected |
| 9 | EXT1 RELAY | Ext 1 in | 25 | N/A |  |
| 10 | N/A |  | 26 | N/A |  |
| 11 | RX BYPASS RELAY | Selects main or RX\_BYPASS\_OUT | 27 | N/A |  |
| 12 | HPF\_BYPASS | RX1 Filter bypass | 28 | HPF\_BYPASS 2 | RX2 filter bypass |
| 13 | N/A |  | 29 | N/A |  |
| 14 | RX MASTER IN RELAY | (selects main, or transverter/ext1) | 30 | N/A |  |
| 15 | REDLED |  | 31 | REDLED 2 |  |

## RX Attenuators

(check whether this is the best way to set them; it might be better to follow protocol 2 if that sets them separately for RX and TX, then only one register set would be needed)

|  |  |  |
| --- | --- | --- |
| **RX Data Conversion Register 1** | | |
| ADC1\_Ctrl[4:0] | ADC1 atten when RX | 5 bit atten setting for RX state; 1dB step |
| ADC1\_Ctrl[9:5] | ADC1 atten when TX | 5 bit atten setting for TX state; 1dB step |
| ADC2\_Ctrl[14:10] | ADC2 atten when RX | 5 bit atten setting for RX state; 1dB step |
| ADC2\_Ctrl[19:15] | ADC2 atten when TX | 5 bit atten setting for TX state; 1dB step |

## TX Attenuators & Drive Level

|  |  |  |
| --- | --- | --- |
| **TX Data Conversion Register** | | |
| DAC\_CTRL[7:0] | RX DAC drive level | PWM DAC drive level when RX |
| DAC\_CTRL[15:8] | TX DAC drive level | PWM DAC drive level when TX |
| DAC\_CTRL[21:16] | RX DAC Attenuation | 6 bit atten value when RX (0.5dB steps) |
| DAC\_CTRL[29:24] | TX DAC Attenuation | 6 bit atten value when TX (0.5dB steps) |

## GPIO register

|  |  |  |
| --- | --- | --- |
| **General Purpose I/O Register** | | |
| GPIO[0] | MIC Bias Enable | =1 to provide electret bias on 3.5mm jack |
| GPIO[1] | Input\_PTT\_Select | 0=PTT on ring; 1=PTT on tip |
| GPIO[2] | Mic\_Signal\_Select | 0=mic on ring, 1 = mic on tip |
| GPIO[3] | Mic\_Bias\_Select | 0=bias on ring; 1= bias on tip |
| GPIO[4] | Spkr\_amp\_Mute |  |
| GPIO[5] | Balanced\_Mic\_Select | =1 to enable balanced mic input |
| GPIO[8] | ADC1 RAND | =1 to randomise data |
| GPIO[9] | ADC1 PGA | =1 to enable ADC 3dB amplifier |
| GPIO[10] | ADC1 DITHER | =1 to dither the clock |
| GPIO[11] | ADC2 RAND | =1 to randomise data |
| GPIO[12] | ADC2 PGA | =1 to enable 3dB amplifier |
| GPIO[13] | ADC2 DITHER | =1 to dither the clock |
| GPIO[15:14] |  | Spare outputs from FPGA |
| GPIO[22:16] | User outputs | Open collector o/p (6 bits) |
| GPIO[24] | MOX (TX strobe) | 1=TX |
| GPIO[25] | TX enable | 1=TX enabled |
| GPIO[26] | (not used) |  |
| GPIO[27] | TX\_Relay\_Disable | 0=normal; 1=TXRX relay & PA disabled |
| GPIO[28] | Puresignal enable | Not used. |
| GPIO[29] | ATU TUNE output | 1=tune |
| GPIO[30] | Transverter enable | 1=transverter |

Some of these signals need gating to provide the required strobes:

|  |  |  |
| --- | --- | --- |
| **Strobe** | **Purpose** | **Logic** |
| MOX | =1 for TX | ((CPU\_MOX || keyer\_MOX) && TX\_ENABLED) |
| DRIVER\_PA\_EN | Controls driver amplifier after DAC. =1 to enable. | Same as MOX |
| CTRL\_TRSW | Additional TX/RX relay | MOX && transverter\_enable |
| TXRX\_RELAY | Controls relay drive into SPI and as an LED output | MOX && ! TX\_RELAY\_DISABLE |

## Status Readback

Not sure about all of these yet!

|  |  |  |
| --- | --- | --- |
| **Status Readback Register** | | |
| **Input Bits** | **Function** | **Meaning** |
| Status[1:0] | PTT in (2 strobes) | 1=TX request  Each PTT is independent; one for 3.5mm jack one for other. |
| Status[3:2] | Key in | 1=key down |
| Status [7:4] | User input 1-4 | User IO4,5,6,8 as drawn  IO5 used as a TX inhibit input  IO8 used as a CW input |
| Status [8] | 13.8v detect in | 1= power valid |
| Status[9] | ATU tune complete | Feedback. |
| Status[10] | ADC1 overflow | Latched overflow indication. |
| Status[11] | ADC2 overflow | Latched overflow indication. |
| Status[15:12] | FIFO prog depth strobes | FIFO nearly empty indications |
| Status[31] | TX\_ENABLE | External input; if 0, TX is gated off |
| Status[63:32] | FPGA ID (4 bytes) | 32 bit user value, holding f/w ID from USR\_ACCESS register |

At the moment the ADC overflow bits are latched, and can be cleared by writing a bit to a register. A better arrangement (which would need new IP) would be to clear the latches when the read operation happens.

# Processor Data Interface

## LED Outputs

Various LED outputs are provided, mostly for debugging. 3.3V logic, LED should connect to ground via a suitable resistor.

|  |  |  |
| --- | --- | --- |
| **LED Output Register** | | |
| **Input Bits** | **Function** | **Meaning** |
| LED\_Out [15:0] | =1 to light LED | To be determined. Initiaally software driven but could be remapped to internal h/w lines. |
| BLINK\_LED | 1Hz blink | Blinks when FPGA configured. |
| PCI\_LINK\_LED | PCIe | Lit when PCIe interface has been initialised by the operating system |

## Data Transfer

From the FPGA hardwire side there are 4 AXI-4 streams of data: two to the hardware (speaker data, I/Q TX data) and 2 from the hardware (microphone samples and multiplexed I/Q samples).

From the processor side there are 3 options for reading and writing data via the PCI express DMA/bridge subsystem:

1. Processor reads and writes via an AXI4-lite interface. This is the easiest to get started and do simple debugging, but may not achieve full performance for protocol 2 I/Q reads. Suggested we use this on data 1 as it will allow rapid progress, but need sot be replaced eventually.
2. DMA reads and writes to separate AXI-4 streams directly interfaced to the IP core. This would be easiest to move across to, once software has been written for the host to manage the DMA engine. It will require that some IP is inserted in the stream to insert TLAST bits every N samples or the DMA engines fail to operate correctly. Smaller FIFO buffers may be acceptable. This approach is probably easiest, but the device driver for ARM processors is VERY slow.
3. DMA reads and writes via an AXI-4 bus interface. Smaller FIFOs may be OK. This can achieve measures 100Mbyte/s over a 64 bit AXI-4 bus but does need IP to access the FIFOs.

Diagram, schematic

Description automatically generated

Figure 15: AXI4 Stream connection to data FIFOs

Diagram, schematic

Description automatically generated

Figure 16: AXI4 Bus connection to data FIFOs

For AXI-4 lite interface to the FIFOs: The choice seems to be to use an AXI streaming FIFO, or to have some simple IP that translates an AXI-4 lite bus transaction to a stream master write (asserting TVALID) or read (accepting TREADY). I have now written suitable IP.

If we used the AXI-4 stream option: DMA transfers over AXI-4 Streams seem to need TLAST asserted to set the transfer size. So for example assert it for every 1024 words written into the FIFO or passed to the DMA engine. Xilinx seems to have example IP to add a TLAST. See here: [Solved: AXI4 Stream - can I fix TLAST to zero and TVALID t... - Community Forums (xilinx.com)](https://forums.xilinx.com/t5/Processor-System-Design-and-AXI/AXI4-Stream-can-I-fix-TLAST-to-zero-and-TVALID-to-one/m-p/590144) search also for **tlast\_gen.v** for example <https://github.com/XavierAudier/tlast_generator>

## FIFO sizes

There are several FIFOs required – see Figure 15:

A picture containing text, calculator

Description automatically generated

Figure 17: CPU to DSP FIFOs

In all cases the FIFOs on the CPU side are 64 bits; the data needs to be resized using AXI stream datawidth converters to match that width.

There 3 cases to consider:

1. RX data: 4 parallel RX streams, variable sample rate
2. TX data: 1 I/Q sample stream, 48KHz (protocol 1) or 192 KHz (protocol 2) sample rate
3. Audio Codec data: 48KHz sample rate, fixed size.

Each block RAM can be 4Kx9, 2Kx18 or 1Kx36 size. (There are many more options!)

### Audio Codec FIFO

This is the simplest. The data set is 16 bits mic samples, 16+16 bits L/R speaker samples at fixed 48KHz rate. At that clock rate a 1K deep FIFO holds 21ms of audio data – far more than required. Separate Codec needed for Mic samples (16 bits wide), L/R speaker samples (32 bits wide)

### TX FIFO

Input data:

* 16 bit I / 16 bit Q samples @ 48KHz Fs (protocol 1)
* 24 bit I / 24 bit Q samples @ 192KHz Fs (protocol 2)

Do the sums for protocol 2; protocol 1 has much lower rate. Work this out for a FIFO width of 4 bytes, and assume that can be read out to unpack the samples.



A reasonable conclusion might be that a 1Kx36 FIFO will be adequate for both protocol 1 and 2 as long as it can be serviced by a new data transfer in <4ms.

### RX FIFO

Input 24 bit I / 24 bit Q samples @ variable Fs. 5 parallel receiver channels; each needs its own FIFO, but DDC0 and DDC1 can be paired (interleaved) to use DDC0 FIFO. In that mode there is an argument for the DDC0 FIFO being larger.

Assume that in protocol 1, the data will be read from the RX hardware in protocol 2 (separated) format then stitched together by software.



The RX FIFOs are the largest memory structure in the design. For processor polling data transfers, the 4K deep FIFO is likely to be needed; for DMA a 1K FIFO may be adequate.

Consider limiting the sample rate on RX5, and giving it a smaller FIFO if needed.

However, this does hinge on the protocol used!

## FPGA – Processor Data Transfer

This is about the organisation of the data. TX I/Q and Codec data are protocol independent; the configuration for receiver samples needs to be considered per protocol.

### TX I/Q data

Regardless of protocol I/Q samples are sent to the TX and these are asynchronous to other transfers. The data is either 16+16 bits @Fs=48KHz (protocol 1) or 24+24 bits @Fs=192KHz (protocol 2). Suggest transfer 24 bits always, and zero pad the LSBs for protocol 1

TX I/Q data requires 24 bit I/Q samples at 192KHz. The same interface needs to be used for the protocol format of 48KHz, 16 bit data after unpacking by the processor. Suggested stick to 24 bit data, with the 16 bit samples left aligned in a 24 bit word (ie zero pad the LSBs). The hardware will be programmable to 48KHz Fs.

The data is 24 bits wide (therefore 48 bits/6 bytes for an I/Q pair). It arrives from Thetis packed into 32 bit words, and hardware unpacking will be required. An AXI-4 Stream data width converter does this simply but there is some ambiguity about the byte positions (particularly as a native processor may be big endian and a PC little endian). Suggested approach therefore is to use a 3 stage process, which uses trivial FPGA hardware:

1. Use an AXI-4 Stream data width converter to expand from 8 to 24 bytes width;
2. Use an AXI-4 Stream subset converter to remap the data bytes as required;
3. Use an AXI-4 Stream data width converter to contract from 24 bytes to 6 bytes.

The processor (or DMA) interface needs to present an AXI-4 stream master interface. For DMA this should be 64 bits wide.

For protocol 1 assume that the ARM processor will unpack the I/Q data with the speaker samples.

### Codec Data

Microphone data requires one stream of 16 bit scalar samples at a sample rate of 48KHz. These will be read by the processor (or DMA engine) directly. The processor (or DMA) interface needs to present an AXI-4 stream slave interface. For DMA this should be 64 bits wide.

Speaker data requires one stream of 16+16 bit Left and Right sample pairs at a sample rate of 48KHz. These will be written by the processor (or DMA engine) directly. The processor (or DMA) interface needs to present an AXI-4 stream master interface. For DMA this should be 64 bits wide.

### RX data, Protocol 1

Protocol 1 interleaves the RX downconverter data and mic data. For the FPGA the RX I/Q streams need to be kept together in case there are any lost packets, because latency of channels for Puresignal will be important. Assume that the processor will merge in the mic samples; the data rate is lower than protocol 2 and when using protocol 1 the processor will only be engaged in data transfers. All DDCs have the same sample rate in this protocol.

In case it helps – there is never any backpressure from the FIFO, so a simple mux is OK.

Diagram

Description automatically generated

Figure 18: RX Data management for Protocol 1

The data format is a sequence of I/Q pairs, multiplexed starting with DDC1. After the 5 I/Q pairs a 16 bit mic sample will be added by the software. The data is split into 64 bit entries in a FIFO after being organised in that manner. 4 sets of samples exactly fits 3 64 bit FIFO words: (H,M,L means high, middle, low byte)

↓byte 0

I1H I1M I1L Q1H Q1M Q1L I2H I2M I2L Q2H Q2M Q2L I3H I3M I3L Q3H Q3M Q3L I4H I4M I4L Q4H Q4M Q4L I5H I5M I5L Q5H Q5M Q5L

### RX data Protocol 2

Protocol 2 keeps each data structure separate, and transferring them from separate FIFOs is the most obvious (and easiest) solution. However it needs to be possible to interleave DDC0 and DDC1. This means there are two conditions that need to be possible (Figure 17). Again, there is never any backpressure from the FIFO.

RX I/Q data requires 24 bit I/Q samples at selectable sample rate 48KHz-1536KHz. Each DDC can have independently set sample rate. The data is 24 bits wide (therefore 48 bits/6 bytes for an I/Q pair). The data needs to arrive in FIFOs that are 64 bits wide.

Diagram

Description automatically generated

Figure 19: RX Data Management (Protocol 2)

For non multiplexed data the required format is:

↓byte 0

I1H I1M I1L Q1H Q1M Q1L I1H I1M I1L Q1H Q1M Q1L I1H I1M I1L Q1H Q1M Q1L I1H I1M I1L Q1H Q1M Q1L (4 I/Q pairs)

For multiplexed data for two DDCs the data will be:

↓byte 0

I1H I1M I1L Q1H Q1M Q1L I2H I2M I2L Q2H Q2M Q2L I1H I1M I1L Q1H Q1M Q1L I2H I2M I2L Q2H Q2M Q2L (2x2 I/Q pairs)

## Parallel I/O

There are a lot of I/O registers!



# IP Modules used in Design

## Xilinx IP

|  |  |  |
| --- | --- | --- |
| **IP** | **Used For** | **Documentation** |
| DMA/Bridge Subsystem for PCI Express | PCI express interface, with DMA and individual read/write capability | PG195 |
| AXI IIC Bus Interface  v2.0 | I2C interface to Codec (IIC is the same thing) | PG090 |
| DDS |  |  |
| FIR |  |  |
| CIC |  |  |

## Local Verilog Modules

Verilog modules used in the design:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Module** | **Inferred attributes** | **RX** | **TX** | **Codec** | **RF Interface** |
| reg\_to\_axis |  | X | X |  |  |
| double\_D\_register | Y | X |  |  |  |
| D\_register | Y |  | X |  |  |
| axis\_mux\_2 |  | X |  |  |  |
| axis\_mux\_4 |  |  | X |  |  |
| regmux\_2\_1 | Y |  | X |  | X |
| regmux\_4\_1 | Y | X |  |  |  |
| regmux\_8\_1 | Y | X |  |  |  |
| axis\_variable |  | X | X |  |  |
| axis\_constant |  |  | X | X |  |
| axis\_adder |  |  |  | X |  |
| i2s\_clk\_lrclk\_gen | Y |  |  | X |  |
| i2s\_xmit | Y |  |  | X |  |
| i2s\_rcv | Y |  |  | X |  |
| cw\_key\_ramp | Y |  | X | X |  |
| debounce | Y |  |  |  | X |
| Serial\_Atten | Y |  |  |  | X |
| SPI | Y |  |  |  | X |
| ClockDivider | Y |  |  |  | X |
| PWM\_DAC | Y |  |  |  | X |
| AXI\_Stream\_Reader |  | X |  | X |  |
| AXI\_Stream\_Writer |  |  | X | X |  |
| AXI\_SPI\_ADC |  |  |  |  | X |
| FIFO\_Monitor |  | X | X | X |  |

# Packaging the Project

This is about getting the FPGA source design into Github.

The approach I have used is:

1. Begin with a simple folder (in my case E:\xilinxdesigns\Pluto)
2. It holds just a few files:
3. Constraints folder – holds the 3 constraints files
4. Sources folder with 3 subfolders:
   1. sources\coefficientfiles – files (generated by spreadsheets) with filter coefficients and keyer waveshape
   2. sources\verilogmodules – HDL sources for the Verilog needed
   3. sources\wrapper – the HDL wrapper which is not automatically managed by Vivado.
5. create\_pluto\_project.tcl: this is a TCL file which reconstructs the project and its block diagram
6. The various git files. .gitignore includes the folder “pluto\_project”

The consequence of this is that git manages the sources files and the TCL script; it does not store all the reconstructed project files. To reconstruct follow the guidance in readme.md:

To use this repository:

1. Install vivado 2020.2

2. Copy this repository to c:\xilinxdesigns\pluto

3. Open vivado and find the TCL command line

4. type: cd c:/xilinxdesigns/pluto

5. type: source create\_pluto\_project.tcl

As the design evolves:

1. create any new Verilog sources in the “sources” folder
2. Periodically recreate the TCL script using the command **File > Project > Write TCL**…
3. Tick “recreate block designs using TCL”
4. Select the “create\_pluto\_project.tcl” file
5. Press OK
6. Publish changes to github



# Linux driver

The XDMA IP core has a Xilinx supplied device driver. See AR65444. Unfortunately it isn’t as simple as it could be. There is a folder missing (/etc/udev/rules.d) and you can get it here: <https://github.com/ramonaoptics/xilinx-dma-driver>

## Building & Patching The Module

The build process requires the files for building kernel modules. The simple way to get them is

**sudo apt install raspberrypi-kernel-headers**

but the documentation says the file could be out of date by several weeks; if the kernel build is recent, you may have to rebuild the kernel from scratch to get them. See <https://www.raspberrypi.org/documentation/linux/kernel/headers.md> and if necessary follow the “build section” link.

The newest code published by Xilinx (with those /etc files added) compile OK on the raspberry pi. However the /dev/xdma0\_user access to axi4-lite bus does not work. Function bridge\_mmap() in file xdma\_cdev.c maps the memory segment: but pci\_resource\_start (around line 196) returns a 64 bit number which is stored into a 32 bit value. Resize the 4 local variables to uint64\_t and it works correctly.

./load\_driver.sh runs OK (need to be root) but doesn’t actually need running as the module loads automatically.

./perform\_hwcount.sh runs, and appears to report success

./dma\_memory\_mapped\_test.sh 1024 16 1 1 runs and reports success

./dma\_streaming\_test.sh 1024 16 1 1 does not run (but there are no streaming DMA channels)

From here: do I need to build the FPGA and download a BIT file? Access the config prom?

In /dev I how have:

/dev/xdma/card0

/dev/xdma0\_c2h\_0

/dev/xdma0\_h2c\_0

/dev/xdma0\_control

/dev/xdma0\_user

/dev/xdma0\_xvc

16 more drivers: /dev/xdma0\_eventsn (/dev/xdma0\_events0 to /dev/xdma0\_events15)

/dev/xdma0\_user is for axi4-lite bus

See instructions in the AR65444 document:

Here is an example of how to read from the bypass channel at a specified offset (0x0000).

$Linux> ./reg\_rw /dev/xdma0\_bypass 0x0000 w

Here is an example of how to write to the bypass channel at a specified offset (0x0000) with specific data (0x1234567): $Linux> ./reg\_rw /dev/xdma0\_bypass 0x0000 w 0x1234567

Application program ‘reg\_rw’ has 32Kbytes allocated space as default. If ‘PCIe to AXI Lite Master’ or ‘PCIe to DMA Bypass’ interface selected size is less than 32Kbytes and try to use ‘reg\_rw’ application for read/write will produce an error. If selected size is less than 32Kbytes modify this define in ‘reg\_rw.c’ to corresponding value and compile (make) the file. E.g. #define MAP\_SIZE (128\*1024UL)

## Data Transfer Performance

Memory mapped reads and writes are limited by the speed of the memory window mapped to the PCIe bus. I seem to achieve approx. 40Mbytes/s write and 4Mbyte/s read with 32 bit transfers. This is OK for register writes but not for data transfer.

The memory mapped DMA performance is transfer size dependent. With the driver recompiled without debug, 4K byte transfers achieve around 50Mbyte/s and 8Kbyte transfers sometimes 80Mbyte/s. Occasional transfers are much slower (1.5us compared with 70us) – possibly a scheduling issue. Data transfer width unknown; it may be possible to speed this up. This is fast enough for dual RX 1536KHz operation, as long as care is taken! The streaming DMA performance is also very slow (maybe 2-5Mbyte/s) so I do need the memory mapped version.

AR68049 is relevant to this!

Data transfers will be from the AXI bus, which can be 64 bits wide. The choice to read an AXI stream from an AXI bus is:

1. Use an AXI streaming FIFO (which presents an AXI bus one one side, but you need to write a transfer length after each batch of accesses); or
2. Create IP to read or write a stream. Pavel Demin’s code used that approach but isn’t readable, so code something similar.

# Raspberry Pi Issues

## LCD DSI port

The “normal” Raspberry Pi boards have a 15 pin, 2 lane DSI connector for the LCD. The CM4 has a 22 pin connector; you can get an adapter or a special flexi PCB with 15 pin at one end and 22 pin at the other.

Need to provide 5V power to the display. Although there is 5V on the flexi PCB, the display does not take power from it. There is a 5 pin 0.1” pitch connector on the display itself with this pinout:

GND; SCL; SDA; (cut off, as an index pin); 5V

The CM4 has 2 display connectors, DISP0 and DISP1. DISP1 is the default, with 4 DSI lanes. There is a documented process to download an overlay (/boot/dt\_blob.bin) for the device tree for the display. As downloaded this drives DISP1. To make it drive DISP0, make these edits in the CM4 section:

DISPLAY\_I2C\_PORT set to <0> (default=0, so no change)

DISPLAY\_DSI\_PORT set to <0> (default=1)

DISPLAY\_SCL set to <1> (default=45)

DISPLAY\_SDA set to <0> (default = 44)

# Things to Do

|  |  |  |
| --- | --- | --- |
| **Item** | **Kit / Environment** | **Description** |
| RX data FIFO | Vivado | Each DDC needs its own data FIFO  Ideally, a multiplexed DDC0/DDC1 FIFO |
| FIFO read/write IP | Litefury | Test the new Verilog IP on the Litefury.  Stretch – see if DMA with 64 bit datapath works. |
| Data transfer documentation | Document | Document the API for data transfers, after doing final tests of FIFO read/write IP |
| SPI ADC i/f | Vivado | Add SPI interface IP to FPGA, and wire up. Pins already allocated. |
|  |  |  |
| ADC overrange detect | Vivado | Change my Verilog IP to add an AXI4-lite bus interface, so it reads data and clears the latch in a single cycle |
| Test FPGA | Vivado, for Saturn | Create a “Test FPGA” block design. ADC1 or ADC 2 routed to DAC; Codec mic input to codec speaker output; a way to test 122.88MHz clock generation; XDMA with some simple registers; I2C connection to config prom; Alex driving code. Essentially enough to do rapid test of h/w once PCB arrives. |
| Litefury FPGA | Litefury | This is to permit writing of data transfer s/w before final h/w available.  Add DDS and datapath FIFOs to simulate RX DDC  Potentially loop back TX FIFO to an RX FIFO  Loop back speaker codec to mic codec  The sample rates need to be nearly right but not perfect. |
| TX datapath | Vivado | Widen TX datapath to 24 bits data |
| Test plan | documentation | Work out how we will test this robustly! |
| Alex Test jig | PCB design | Make a simple “alex emulator” with LEDs for each data bit |
| DSP performance | Vivado | Improve DSP performance to that required for final radio. Likely to need a working connection to pihpsdr or Thetis. |
| Pi test data transfer s/w | Pi / litefury | Write code to manager reading or writing one FIFO using DMA, so we achieve a constant stream. |
| Pi protocol 2 | Pi / litefury?? | Write a complete protocol 2 data transfer app. If I can emulate the DDC & DUC well enough, we might be able to do this using the litefury before final h/w is available. |
| IDE / s/w tools | Pi | Find more comprehensive software tools. IDE for editing, integrated debug. |
| Desktop github | Pi | Find linux equivalent of github desktop |
| Github repository | Pi/ PC | Commit repository to github |
| Refactor block design into IP | Vivado | Create formal IP blocks for DDC, DUC, CW keyer, codec interface. Probably tis only makes sense after the functions are substantially complete. |
| pihpsdr | pi | Modify pihpsdr to have direct data transfers, allowing a “front panel” radio operation in the same RPi |
| Config prom app | pi | Write a GUI app to write the config prom, replacing the current command line version |

# Useful Information

Phil Harman provided this post about PWM: [[hpsdr] EER Support in PowerSDR and Hermes (openhpsdr.org)](http://lists.openhpsdr.org/pipermail/hpsdr-openhpsdr.org/2014-June/045625.html)

A list of linux IDE tools came up with some candidates to look at. Each claims to have integrated tool chain and debugger.

|  |  |  |
| --- | --- | --- |
| **Order to try** | **IDE** | **Comment** |
| 1 | Eclipse CDT |  |
| 2 | K Develop |  |
| 3 | Qt Creator |  |
| 4 | Oracle Solaris Studio |  |
| 5 | Codelite |  |
| 6 | Code::Blocks |  |
| 7 | Anjuta |  |
| 8 | Gnat Programming Studio |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

See: [21 Best Free Linux Integrated Development Environments (IDEs) - LinuxLinks](https://www.linuxlinks.com/ide/)

See: [Comparison of integrated development environments - Wikipedia](https://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments#C/C++)

# Notes

* I removed 2 strobe signals from the original Andromeda design:
  + Puresignal\_Enable (which should never have been a strobe – it is used in Orion purely to choose which frequency is used in DDC4)
  + Bias\_Ctrl(which was never an Orion strobe);
* TXRX\_Relay is retained as a strobe, to drive an LED when TX is asserted & enabled

1. Source: Xilinx AR# 51017 [↑](#footnote-ref-1)
2. There appears to be no constraint between GTP supplies and VCCAUX & VCCO: so they could be powering up while MGTAVCC and MGTAVTT power up. [↑](#footnote-ref-2)