# Protocol 2 Data Transfer

# Byte Ordering

The network ports use “network byte ordering” with all values being high byte first. On both the PC and ARM processors, the natural byte ordering for processor reads and writes is Low byte first so you can’t simply write or read words.

Linux provides standard functions to convert:

|  |  |
| --- | --- |
| uint32\_t htonl(uint32\_t arg) | Convert 32 bit value from host to network format |
| Uint16\_t htons(uint16\_t arg) | Convert 16 bit value from host to network format |
| uint32\_t ntohl(uint32\_t arg) | Convert 32 bit value from network to host format |
| Uint16\_t ntohs(uint16\_t arg) | Convert 16 bit value from network to host format |

There is a mechanism in **General Packet to SDR** to change the byte ordering. Saturn doesn’t yet support this – I’d need programmable byte swaps on I/Q and audio data. Not sure if Thetis supports it either.

FPGA registers have the same byte ordering as the host as seen through the PCI Express data transfer. You can use register reads and writes directly without conversion.

# Port Numbers to Use

Discovery packet comes to port 1024 in SDR. Reply to the port number in that message.

Then the PC sends a General Packet to the SDR (p19) with port numbers to use. If values given are zero, use the defaults here. The values coming from Thetis match the defaults anyway. The code will have a table of port & socket information for the various threads, with the table index given below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Port number in PC** | **Port number in SDR** | **Index** |
| Program |  | 1024. not supported. | 0 |
| Erase |  | 1024. not supported. | 0 |
| Set IP Address |  | 1024. not supported. | 0 |
| Command reply | Source port in command message |  |  |
|  |  |  |  |
|  |  |  |  |
| Discovery |  | 1024 | 0 |
| Discovery reply | Source port in discovery message  (also says 1024 in spec) |  |  |
| General packet to SDR |  | 1024 | 0 |
| DDC Specific Port |  | 1025 | 1 |
| DUC Specific Port |  | 1026 | 2 |
| High Priority from PC |  | 1027 | 3 |
| High Priority To PC | 1025 |  | 6 |
| DDC Audio (speaker) |  | 1028 | 4 |
| DUC0 I/Q data |  | 1029 | 5 |
| DDC0 I/Q data  DDC1 I/Q data  DDC2 I/Q data  DDC3 I/Q data  DDC4 I/Q data  DDC5 I/Q data  DDC6 I/Q data  DDC7 I/Q data  DDC8 I/Q data  DDC9 I/Q data | 1035  1036  1037  1038  1039  1040  1041  1042  1043  1044 |  | 8  9  10  11  12  13  14  15  16  17 |
| Mic samples | 1026 |  | 7 |
| Wideband ADC0  Wideband ADC1 | 1027  1028 |  | 18  19 |
| Memory mapped from PC |  | Undefined (not supported) |  |
| Memory mapped to PC | Undefined (not supported) |  |  |

We’ll need to request a change to the protocol 2 specification. We can have sockets bound to ports ready to receive messages in advance. But if the general packet to SDR changes the port numbers, the sockets will have to be closed down and new ones opened and re-bound. Thetis sends its next 3 messages out immediately after the general packet to SDR. By the time that’s done, the messages will have been missed and you get “message rejected” indications in wireshark. Proposed change: if non standard ports numbers are used, have a 20ms delay before further messages.

All outgoing messages go to the same PC port, as used in the discovery request. If it is the **from** address set to these values.

# Code operation

The code strategy in response to the initial message sequence is:

|  |  |  |
| --- | --- | --- |
| **PC sends message** | **SDR sends message** | **Notes** |
| Startup | | |
| **P1 discovery**  **P2 discovery** |  | (note this is also to port 1024) |
|  | **P2 reply** |  |
| **General Packet** |  | This provides the required port numbers; ports can’t be bound until this happens.  SDR must establish listeners on the relevant ports. 5 “listener” threads established, each provided with a pre-configured port.  Outgoing threads also established, with a link to the general command port for message sends. |
| **DDC Specific** |  | (process and action as normal) |
| **DUC Specific** |  | (process and action as normal) |
| **High Priority**, run bit=1 |  | “run” bit is set in the first of these.  Assert DataThreadsEnabled  Outgoing data threads commence sending data.  (process the remainder as normal) |
| Normal operation | | |
| **DDC Specific,**  **DUC specific** |  | Sent if retuned |
| **High priority** |  | Sent if any changes |
|  | **High priority** | Sent every 1ms (in TX) or 50ms (in RX) |
| **DUC samples**  **Spkr audio** | **DDC samples**  **Mic samples** | At normal rates. Note several threads for DDC samples, depending on configuration; so we need some run-time setting of options. |
| Shutdown | | |
| **High Priority**, run bit=0 |  | Deassert DataThreadsEnabled  “listener” threads close their ports & terminate  Outgoing threads just terminate |

At the Raspberry pi end, I need to open ports with “listener” threads as shown in the 3rd column immediately below after getting the General Packet. That suggests I need 6 read threads. The first will be the main “p2app” threat that will hand the command messages in port 1024. The others need to be initiated after the general packet has arrived.

Note that there is no “run/stop” packet. That information is embedded into a high priority data packet; so its arrival will need to signal the outgoing data threads.

|  |  |  |
| --- | --- | --- |
| **Thread** | **Purpose** | **Port** |
| Main(); | RX & process **Discovery Packet**  Send **Discovery Reply Packet**  RX & process **General Packet to SDR**  Spin up the new threads when **General Packet to SDR received** with port numbers | Rx at: 1024 |
| IncomingDDCSetup(); | RX & process **DDC Specific Packet** | Rx at: 1025 |
| IncomingDUCSetup(); | RX & process **DUC Specific Packet** | Rx at: 1026 |
| IncomingHighPriority(); | RX & process **High Priority from PC Packet** | Rx at: 1027 |
| IncomingDUCIQ(); | RX & process **DUC I/Q Data Packet** | Rx at: 1029 |
| IncomingSpkrAudio(); | RX & process **DUC Audio Packet** | Rx at: 1028 |
| OutgoingHighPriority(); | Generate **High Priority to PC Packet**. Every 50ms(when in RX) or every ms (in TX). | To: 1025 |
| OutgoingMic(); | Read Mic samples FIFO and generate **Mic Samples Packet** | To: 1026 |
| OutgoingDDCIQ(); | Read I/Q FIFO for DDC or interleaved DDC pair. **Generate DDC I/Q packet**.  Ideally one function, re-used to set up several threads | To: 1035-1044 |

DDC reconfiguration is done while running “live” so we need some dynamicness. Probably means the outgoing threads need to be active, and take commands. Changeover of interleaved/non interleaved DDC may need specific actions.

Begin by coding the main() loop, to receive discovery & general packets (with stubs for the packets we don’t use) and spin up the other threads. Implement the on/off functionality to get the minimal data transfer before generalising.

Incoming RX threads need no specific start/stop action. The far end will just stop sending data. They do need to be able to shut down and re-initialise a socket with a new port number.

Outgoing threads need the ability to be given commands:

* Close socket and re-open with a new port number
* Start/Stop data transfer
* DDC threads: Change between interleaved and non interleaved DDC while still running

We have a command word to each thread, with the intent that bits will be used for each possible command.

# DMA Transfers

IQ and Codec audio data needs to be transferred between the FPGA and the Raspberry pi host processor. The only efficient way to do that is using DMA transfers, using the DMA engines built into the FPGA PCI express interface. There is a Linux device driver for them, and (after minor tweaks) they work in both 32 and 64 bit Linux.

DMA transfers are memory-memory. At the FPGA end, they access FIFOs. A FIFO only has one address, but the interface to the accepts a wide address range so you can do a DMA with incrementing address pointer. Transfer speeds are heavily influences by setup and shutdown times; a 4Kbyte transfer seems to take ~100us implying c.40Mbyte/s. There are two DMA engines in each direction (read from FPGA, write to FPGA) that can be used concurrently. We will need to match the transfer size to the expected data rate.

There are only two data sources that need to be DMA written to the FPGA (I/Q data and codec speaker data) so each can have its own DMA. However I/Q data and mic samples will need to share two DMA registers.