Image streaming UDP protocol

Julian Tilbury
Plankton Analytics ltd.

Introduction

The Pi_Imager sends image data as UDP datagrams over a local area network on port 5000 of the designated network address (see PiImagerConfig.txt, in code example 1). The example below shows loopback (127.0.0.1) allowing the receiving software to run on the same computer as Pi_Imager for testing. The default address broadcast address is shown also. Please note that UDP does not have any error correction mechanism, so files are sent assuming a good quality connection.

(a) Test:

```
ImageBroadcast {
  IpAddress     = 127.0.0.1
  Port          = 5000
}
```

(b) Default:

```
ImageBroadcast {
  IpAddress     = 192.168.0.255
  Port          = 5000
}
```

Code example 1: UDP packet broadcast address as specified in PiImagerConfig.txt on PiPC (a) test, (b) default)

Data Format

Files are send as a sequence of (up to) 8216 byte packets, consisting of a 24 byte header and (up to) 8k data. A tiff file is sent as at least 3 packets. The first packet is the filename (including its file path) , the second is the
tiff file header, the third and subsequent packets are the image data. If a packet doesn't need 8k, (the first and second packet invariably don't) the packet is shorter. Many images are likely to be less than than 8k bytes.

The tiff header, as sent in the second packet, is exactly what would be written to disk for a normal tiff file. It was designed this way, so a tiff file is written to disk by just ignoring the first packet and writing the rest raw to disk (taking account of the data lengths of each packet).

The 24 byte packet header is:-

Bytes	Name	Type	Description
0 ... 3	Hash	unsigned 32 bit integer	A hash code of the packet, being a sum of the unsigned bytes from (and including) UniqueId to the end of the data. This is a shite

hash code, that was a place holder for something better later.

4 ... 5 FileIdx unsigned 16 bit integer - The number of the file. All packets of the same file have the same FileIdx.

PI_Imager starts from 0, and increments to 2047 (check!) then restarts at 128 (check!), counts up to 2047, are cycles back to 128. That way, if Pi_Imager is suddenly stopped and started any files flying through the ether with file Idxs 0 to 127 are from the restart - they can't clash, even if the old invocation had just ticked past 2047

6 ... 7 PartIdx unsigned 16 bit integer - The number of the packet in the file. Starts from 0. If a tiff file, PartIdx 0 will always be the filename packet, PartIdx 1 the tiff header, PartIdx 2 the first data packet etc.

8 ...15 UniqueId unsigned 64 bit integer - A unique file id based on a time stamp of nanoseconds since 1970. Good

for 1970 to 2323. Due to the coarse granularity of the clock, if more that one image is produced per clock tick, the UniqueId is incremented by one for each file. It is not a timestamp, it is a unique file Id.

16 ...17 TotalParts unsigned 16 bit integer - The total number of packets in a file. If TotalParts is 4, there should

be FileIdx 0 - filename, FileIdx 1 - tiff header, FileIdx 2 first image data packet, FileIdx 3 second image data packet.

18 ... 19 DataSize unsigned 16 bit integer - The number of bytes in the data part of the packet. If DataSize is 1024

the packet should be 1048 bytes long, consisting of the 24 byte header and 1024 bytes of data. Max packet is 8K. min packet is 1 (zero length data shouldn't be sent)

20 ... 21 Tag unsigned 16 bit integer - A numeric code for the type of a packet:-

0 NoTag - shouldn't be sent
1 Filename - the filename (first packet of a file)
2 TiffIfd - a tiff header (second packet of a file)
3 FileBody - ordinary file data (not a tiff file)
4 TiffBody - tiff file image data

22 ... 24 Packing unsigned 16 bit integer - A couple of padding bytes that should always be zero.

As implied by a Tag of 3 (FileBody) ordinary files, for instance the report file Cameralog.txt, are also transferred by this protocol. In this case, the first packet is the filename, the subsequent packets are the data.

The filename is given relative to the SurveyDirectory - the directory the user sets up on Pi_Imager to hold all runs. It will contain a list of day directories, each containing 10 minute directories. So the filenames will be something like 2023-01-31\1030\RawImages\pia1.2023-01-31.1030.N00000000.tif. It is designed so that this filename can be easily appended to the archive directory root on the receiving machine to

reconstruct the survey directory structure . E.g. C:\Archive\SpringSurvey + \ + “2023-01-31\1030\RawImages\pia1.2023-01-31.1030.N00000000.tif” (only the part in quotes is in the packet).
Directories are not sent. The receiving program must pay attention to the file paths it is being sent and create directories as appropriate. Given UDP protocol doesn't guarantee packet order or even delivery, sending directories would be useless as they might arrive after the file that required them, or not at all.

The height and width of the tiff image is buried in the tiff header. We have C++ code to efficiently extract it, but given it is a standard tiff header, Python libraries should be able to read it. They probably expect it from a file on disk though, not in a sequence of bytes in a UPD packet.

The images are sent raw, and should be intensity corrected cf. the average background. Currently the average background is sent as an extra line of data, So a width of 100, by height of 80, image will be sent as 8100 bytes, but the tiff header will still give the height as 80. In future, we will probably do the intensity correction in PI_imager, especially in light of this project.

There is some additional information, in a proprietary format, "hidden" in the tiff header. But to any standard tiff reader it will look like an ordinary tiff header.

The protocol was designed for fast processing of UDP packets which can arrive out of order. There is an array of 2048 file records allocated at boot, a large ring buffer of pointers to packets, and a stack of pointers to packets, pointing to a large array of packets, also allocated at boot.

As each file has a FileIdx, it already knows its place in the file array. All packets of that file carry the number of packets in the file, so as soon as any packet is seen its size can be recorded, and a sequence of place reserved in the ring buffer. The pointer to the current buffer is assigned to the appropriate slot in the ring buffer, and the current buffer pointer replaced from one popped off the stack, ready to receive the next packet. When a file (eventually) receives all its packets (simple countdown), the packets are written to disk, the packet pointers pushed back on the stack, and the file record cleared. The only complexity is dealing with the multiple error conditions when packets are late or missing, or have bad (check) data, and then tidying up the data structures as appropriate.

