Pixirad-1 Read-Out data structure

Measurement data comes out the Pixirad-1 fragmented in a set of 360 (135 for Autocal Data) UDP packets 1448 bytes each. Table 1 summarizes the datagram structure. The receiver client can recover from the packet header informations for measurement data subsets resembling.

UDP Packet Structure			
Offset(bytes)	Length(bytes)	Name	More
0	2	PACKET_TAG	0:PACKET_TAG
			1:SLOT_ID
2	2	PACKET_ID	0:PACKET_ID_LSB
			1:PACKET_ID_MSB
4	1440	*Counters Data*	
1444	4	*Future Use*	

Table 1: UDP packet header structure

Table 2 gives the bit assigment in PACKET_TAG byte.

-PACKET_TAG byte can be used by the client to divide Autocal data from measurement data. Bit 5 in PACKET_TAG byte is used to propagate to others tasks (monitor) alerts on data reliability.

-PACKET_ID is 2 bytes long and contains an incremental numer ranging from 0 for first data subset to 359 (134 for Autocal data) for the last. For Pixirad-1 DAQ firmware revisions earlier than “Dec2013” this number is actually modulo 45.

-SLOT_ID stores a frame identifier that increments over Images transmission. Every UDP packet belonging to the same image would have the same SLOT_ID.

PACKET_TAG byte Structure		
Bit	**Function**	**Read/Write**
7	Register0/1	Read
6	Autocal Data	Read
5	Frame Has Aligment Errors	Write
4	TBA	
3	TBA	
2	TBA	
1	TBA	
0	TBA	

Table 2: PACKET_TAG Bit Functions

Table 3 provides an example on counter data arrangement in UDP packet payload. Assuming that readings start at offset 4 the example wouls refer to the following counter conversion:

Counter 0 in cluster 0 counts 1
Counter 0 in cluster 1 counts 1
Counter 0 in cluster 2 counts 1
…....
Counter 0 in cluster 13 counts 1
Counter 0 in cluster 14 counts 2
Counter 0 in cluster 15 counts 1

Counter 1 in cluster 0 counts 2
Counter 1 in cluster 1 counts 1
Counter 1 in cluster 2 counts 1
…....
Counter 1 in cluster 13 counts 3
Counter 1 in cluster 14 counts 1
Counter 1 in cluster 15 counts 3

Please notice that counters implement a pseudo random counting scheme. Pixels data are readout from top to bottom for odd columns and fro bottom to top for even columns. Data

Unsigned short sequence	Counter Matrix Cluster																
US-offset	Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
15	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	27	**2**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
	28	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**
	29	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

Table 3:Counters data Arrangement example

A measurement data client should collect all the expected UDP packets belonging to an image and resemble them in a bufter after header and trailer has been removed. In the following some pseudo (actually not so “pseudo”) C code is provided to summarize the data image reconstruction. It is the “udp_client.exe” image reconstruction core. It works on the assumption that process_buff_ptr” is an Unsigned Short buffer long enough to contain an entire image (512x476) and 2 bytes that replicates PACKET_TAG received in UDP datagrams.

```
#define PIXIE_COLS                              512
#define PIXIE_ROWS                              476
#define PIXIE_DOUTS                             16
#define MATRIX_DIM_WORDS                    (PIXIE_COLS*PIXIE_ROWS)
#define PIXELS_IN_LAST_SECTOR               15232
#define PIXELS_IN_LAST_SECTOR_MAP           15232

#define SECTORS_IN_PIXIE                    16
#define COLS_PER_DOUT                       32
#define COLS_PER_DOUT_LAST_SECTOR           32


#define PACKET_TAG_BYTES 2
#define PACKET_TAG_OFFSET 0
#define PACKET_ID_BYTES 2
#define PACKET_ID_OFFSET 2

#define FRAME_HAS_ALIGN_ERRORS 0x20
#define REG_PACKET 0x80
#define SLOT_ID_MASK 0xff
#define SLOT_ID_OFFSET 1
#define AUTOCAL_DATA 0x40
#define AUTOCAL_REG_DEPTH 5
#define COUNTER_REG_DEPTH 15



int is_autocal_data, reg_data,i,j,k,code_depth,err;
unsigned short *local_buffer_ptr,*temp_us_ptr,*conv,*process_buf_ptr;
unsigned short packet_tag,slot_id=0;
unsigned short this_frame_has_aligment_errors;



local_buffer_ptr=databuffer_allocation(MATRIX_DIM_WORDS*15)
conv=conversion_table_allocation();



packet_tag=*(process_buf_ptr+PACKET_TAG_OFFSET*2);
slot_id=*((char*)process_buf_ptr+SLOT_ID_OFFSET)&SLOT_ID_MASK;

if(packet_tag & AUTOCAL_DATA){
            code_depth=5;
            is_autocal_data=1;
            }

else{
            code_depth=15;
            is_autocal_data=0;}

if(packet_tag & REG_PACKET)
      reg_data=1;
else
      reg_data=0;
```

```
if(packet_tag & FRAME_HAS_ALIGN_ERRORS)
      this_frame_has_aligment_errors=1;
else
      this_frame_has_aligment_errors=0;


temp_us_ptr=process_buf_ptr+(PACKET_TAG_BYTES/2);

for(j=0;j<COLS_PER_DOUT*PIXIE_ROWS;j++)
      for(k=0;k<code_depth;k++){
      my_bytes_swap(temp_us_ptr+i+(j*code_depth)+(k));
      local_buffer_ptr[(i*COLS_PER_DOUT*PIXIE_ROWS*code_depth)+
      (j*code_depth)+k]=temp_us_ptr[i+(j*code_depth)+(k)];
      }

for(j=0;j<COLS_PER_DOUT*PIXIE_ROWS;j++)
convert_bit_stream_to_counts(
      code_depth,
      local_buffer_ptr+(i*COLS_PER_DOUT*PIXIE_ROWS*code_depth)+(j*code_depth),
      process_buf_ptr+(i*MATRIX_DIM_WORDS)+(j*PIXIE_DOUTS)+(PACKET_TAG_BYTES/2),
      PIXIE_DOUTS);

if(is_autocal_data==0 && convert_data==1)
                  decode_pixie_data_buffer(
                        conv,
                        process_buf_ptr+(PACKET_TAG_BYTES/2)+i*MATRIX_DIM_WORDS);
databuffer_sorting(process_buf_ptr+(PACKET_TAG_BYTES/2)+i*MATRIX_DIM_WORDS);
map_data_buffer_on_pixie(process_buf_ptr+(PACKET_TAG_BYTES/2)+i*MATRIX_DIM_WORDS);
```

Code Segment 1

Code Segment 1gives the flow of operations to be performed on the raw image buffer data to get the corresponding Image. In the following code segments are given for customs function called in Code Segment 1.
Code Segment 2 list the code used to allocate and fill the conversion table needed to get counters binary natural code from the psedudo random sequence. Code Segment 3 lists all the functions used in Code Seg.1 to resemble data converting the bit stream to counters matrix and to map them on the physical pixel matrix.

```
#define  PSCNT_WIDTH    15
#define  PSTABLE_DEPTH  32768

void genera_tabella_clock(unsigned short *clocks, unsigned short dim, unsigned
short counterwidth){
            //unsigned int clocks[32768];
            unsigned long potenze[16], bit1,bit2;
            unsigned long i,tempo;


            potenze[0]=1;
            for (i=1; i<counterwidth+1; i++)
                  potenze[i]=potenze[i-1]*2;

            clocks[0]=0;
            tempo=0;
            for(i=1; i<dim; i++){

                  bit1=tempo&potenze[14];
                  if(bit1 != 0)bit1=1;
                  bit2=tempo&potenze[6];
                  if(bit2 != 0)bit2=1;
                  bit1=!(bit1^bit2);
                  tempo=tempo*2+bit1;
                  tempo=tempo%potenze[15];
                  clocks[tempo]=i;

            }
            clocks[0]=0;
            return;
      }

unsigned short * conversion_table_allocation(void){
      unsigned short *ush_ptr;
      ush_ptr=(unsigned short*)calloc(PSTABLE_DEPTH, sizeof(unsigned short));
      if (ush_ptr!=NULL)
            genera_tabella_clock(ush_ptr,PSTABLE_DEPTH,PSCNT_WIDTH);
      return(ush_ptr);
```

Code Segment 2

```
int my_bytes_swap(unsigned short* us_ptr){
      char a,b,*temp_char_ptr;
      temp_char_ptr=(char*)us_ptr;
      a=*temp_char_ptr;
      b=*(temp_char_ptr+1);
      *(temp_char_ptr+1)=a;
      *(temp_char_ptr)=b;
}

int convert_bit_stream_to_counts(
                        int code_depth,
                        unsigned short* source_memory_offset,
                        unsigned short* destination_memory_offset,
                        int resulting_readings){
      int i,j;
      unsigned short dout_masks[resulting_readings],mask_seed=1;
      for(i=0;i<resulting_readings;i++) dout_masks[i]=(mask_seed<<i);
      for(j=0;j<resulting_readings;j++){
            destination_memory_offset[j]=0;
             for(i=code_depth-1;i>=0;i--){
                  if(source_memory_offset[i] & dout_masks[j])
                        destination_memory_offset[j]|= dout_masks[code_depth-i-1];
                  else
                        destination_memory_offset[j]&= ~dout_masks[code_depth-i-
1];
                  }
            }
      return(j);}


void decode_pixie_data_buffer(unsigned short* table, unsigned short* databuffer){
      int i;
      for(i=0;i<MATRIX_DIM_WORDS;i++)
            databuffer[i]=table[(0x7fff)& databuffer[i]];
      }

int databuffer_sorting(unsigned short *buffer_a){
      extern unsigned short  grouped_cols;
      unsigned short PIXELS_IN_SECTOR;
      unsigned short    *buffer_b;
      unsigned  long sector_cntr,pixel_cntr;
      PIXELS_IN_SECTOR=(COLS_PER_DOUT * PIXIE_ROWS);
      buffer_b=(unsigned short*)calloc(
                                          PIXIE_COLS*PIXIE_ROWS,
                                                sizeof(unsigned short));
      if (buffer_b==NULL) {
                  printf("\r\nDATA sorting:Memory allocation unsuccesfull!!
                        Please contact an expert :-)");
                  return(0);}

      for(sector_cntr=0;sector_cntr<SECTORS_IN_PIXIE-1;sector_cntr++){
       for(pixel_cntr=0;pixel_cntr<PIXELS_IN_SECTOR;pixel_cntr++){
            buffer_b[((sector_cntr+1)*PIXELS_IN_SECTOR_MAP)-pixel_cntr1]=
            buffer_a[sector_cntr+pixel_cntr*SECTORS_IN_PIXIE];}
            }
```

```
      for(pixel_cntr=0;pixel_cntr<PIXELS_IN_LAST_SECTOR;pixel_cntr++){
      buffer_b[((sector_cntr)*PIXELS_IN_SECTOR_MAP+PIXELS_IN_LAST_SECTOR_MAP)-
      pixel_cntr-1]=buffer_a[sector_cntr+pixel_cntr*SECTORS_IN_PIXIE];}

      //copying buffer to the original one

      for(pixel_cntr=0;pixel_cntr<(MATRIX_DIM_WORDS);pixel_cntr++)
            buffer_a[pixel_cntr]=buffer_b[pixel_cntr];
      free(buffer_b);
      return(1);
      }


//map_data_buffer_on_pixie rearranges data in PIXIE layout taking in account the
"snake" readout architecture
int map_data_buffer_on_pixie(unsigned short *buffer_a){
      unsigned short* temp_col;
      unsigned short col_cntr,row_cntr;

      temp_col=(unsigned short*)calloc(PIXIE_ROWS, sizeof(unsigned short));
      if (temp_col==NULL) {
                        printf("\r\nDATA mapping:Memory allocation unsuccesfull!!
P                             lease contact an expert :-)");
                        return(0);}
      for(col_cntr=0;col_cntr<PIXIE_COLS;col_cntr++){
            if ((col_cntr%2)){//only odd index columns are reversed
                  for(row_cntr=0;row_cntr<PIXIE_ROWS;row_cntr++){
                  temp_col[PIXIE_ROWS-row_cntr-1]=
                  buffer_a[col_cntr*PIXIE_ROWS+row_cntr];}

                  for(row_cntr=0;row_cntr<PIXIE_ROWS;row_cntr++){

      buffer_a[col_cntr*PIXIE_ROWS+row_cntr]=temp_col[row_cntr];}
            }
      }
      free(temp_col);
      return(1);}
```

Code Segment 3

All the given code pieces are ready to compile.

Unpdates from previous versions	
Feb2014.1.2	*"**Slot_ID**" in UDP packet payload has been made avalilable*