Software Interfaces for R2DAQ

Document Version: 0.1.61

André Young

November 19, 2015

1 Introduction

This document describes software interfaces for the ROACH2-based data acquisition system developed for Project 8.

In Section 2 an overview of operating the ROACH2 is given. This is followed by a detailed description of the software interfaces for controlling the ROACH2 system in Section 3. Section 4 describes the data capturing software that runs on the compute server.

2 Overview of ROACH2 Operation

The run-cycle of the fielded system will typically proceed as follows.

1. Initialization
   (a) Power-up ROACH2[1]
   (b) Open connection
   (c) Start bit-code
   (d) Calibrate ADC modules
   (e) Initial configuration[2]
2. Run
   (a) (Re-)Configuration[2]
   (b) Slow monitoring[2][3]
   (c) Data capture
      i. Fast monitoring[2]

The system will be powered up and initialized once, and then set to run continuously, possibly with configuration updates from time to time. Reconfiguration will usually result in a brief interruption in the data stream. Slow monitoring (metadata and data) is aimed at verifying correct system configuration prior to capturing data, whereas fast monitoring (metadata only) is typically used to monitor the system continuously while capturing data.

A description of software tools available for some of the above aspects is described in the following.

[1] Provided system configured as "always on", i.e. whenever AC power is available the system will power up.
[2] Read from and/or write to software registers accessible within the bit-code.
[3] May include accessing and analyzing streamed data.

3 Control / Monitoring Interface

Control and monitor interfaces are mainly built on the `corr` python package, which uses KATCP (Karoo Array Telescope Control Protocol) for communicating with the ROACH2 system. The ROACH2 system is also accessible via SSH for lower-level control. These communications are all routed over a 1 GbE connection to the ROACH2.

For monitoring it is often useful to also perform diagnostic analysis on the data output by the system. Access to data is possible either via buffering short frames and reading over 1 GbE, or by grabbing packets from the main data stream (i.e. multiple 10 GbE interfaces).

The ROACH2 is also accessible via USB. This interface is mainly used for once-off configuration of a newly assembled system.

3.1 Connecting to and programming bit-code on a ROACH2

This is the first step needed after powering up a fielded system. The code listing below[4] shows how to connect to a ROACH2 and program it with a given bit-code.

Listing 1: Connecting to and programming bit-code on a ROACH2.

```
# python package to communicate with roach2
import corr

# connect to roach2 using its network name
r2daq = corr.katcp_wrapper.FpgaClient('roach2-hostname')
r2daq.wait_connected()

# program a bit-code
bof_to_load = 'r2daq_bitcode_v0.7.bof'
bof_list = r2daq.listbof()
if not bof_to_load in bof_list:
        # if bit-code not yet uploaded, do this first
        upload_port = 7147
        r2daq.upload_bof(bof_to_load,upload_port)
r2daq.progdev(bof_to_load)
r2daq.wait_connected()
```

3.2 ADC calibration

The library adc5g is available for general testing and calibration of the ADCs on the ROACH2.

Calibrating the ADCs should be performed after a suitable bit-code has been programmed (i.e. a bit-code that uses the CASPER "yellow blocks" for the ASIAA 5 Gsps ADC). The code listing below shows how to do this.

Listing 2: Calibrating the ADCs.

```
# python package with adc utilities
import adc5g

# set adc in test mode
adc5g.set_test_mode(r2daq,0)
adc5g.set_test_mode(r2daq,1)
# calibrate and print diagnostic output
adc5g.sync_adc(r2daq)
```

[4] In the interest of not repeating similar sections of code subsequent listings assume that earlier listings have been executed within the same scope.

```
opt, gls = adc5g.calibrate_mmcm_phase(r2daq,0,['r2daq_snap_8bit_0_data',])
print opt, glitches
opt, gls = adc5g.calibrate_mmcm_phase(r2daq,0,['r2daq_snap_8bit_0_data',])
print opt, glitches
# unset adc test mode
adc5g.unset_test_mode(r2daq,0)
adc5g.unset_test_mode(r2daq,1)
```

3.3 Other initial configuration

After loading a specific bit-code some initial configuration is usually needed.

For example, each of the used 10 GbE interfaces may need to be configured, e.g. assigned IP and MAC addresses. The code listing below shows how to do this, assuming that data is sent over the connection `192.168.1.100:6000 --> 192.168.1.2:6001`, from the ROACH2 to interface `eth3` on the compute server that executes this code.

Listing 3: Configuring a 10 GbE interface.

```
# general network interface utilities
import netifaces as ni

# use locally administered mac, e.g. 02:xx:xx:xx:xx:xx
mac_addr = (2<<40) + ip_addr
# set src ip address value
ip_str_src = "192.168.1.100"
ip_split = [int(ch) for ch in ip_str_src.split('.')]
ip_addr_src = 0
shift = 24
for x in ip_split:
        ip_addr_src = ip_addr_src + (x<<shift)
        shift = shift - 8
# set src port value
port_src = 6000

# set dest ip address value
ip_str_dest = "192.168.1.2"
ip_split = [int(ch) for ch in ip_str_dest.split('.')]
ip_addr_dest = 0
shift = 24
for x in ip_split:
        ip_addr_src = ip_addr_dest + (x<<shift)
        shift = shift - 8
# set dest port value
port_dest = 6001

# setup arp table, entry arp[x] associates with ip 192.168.1.x
arp = [0xffffffffffff] * 256
# this gets the mac address of interface which will receive data stream
mac_eth3 = ni.ifaddresses('eth3')[17][0]['addr']
mac_eth3 = int(mac_eth3.translate(None,':'),16)
arp[2] = mac_eth3

# configure 10gbe core (src)
r2daq.config_10gbe_core('r2daq_tx0_core',mac_addr_src,
        ip_addr_src,port_src)
# configure 10gbe core (dest)
```

```
r2daq.write_int('r2daq_tx0_dest_ip',ip_addr_dest)
r2daq.write_int('r2daq_tx0_dest_port',port_dest)
```

3.4 Application specific configuration

Application-specific configuration of the ROACH2 system typically requires reading from / writing to software registers available from within the bit-code.

The names and functions of registers are not yet determined, but for example let a register named `r2daq_digital_f0` be 32-bit wide and let the lower thirty bits be subdivided onto three 10-bit words that each define the center frequency of a digital channel selected within the IF. The following code listing shows how the current configuration may be read, and then updated.

Listing 4: Reading and writing configuration registers.

```
from numpy import arange, float32
# mapping from 10-bit integer to center frequency in MHz
freq_list = arange(0.0,3200.,3200./1024.,dtype=float32)

# print current configuration
digital_f0 = r2daq.read_int('r2daq_digital_f0')
ch0_f0 = freq_list[digital_f0 & 1023]
ch1_f0 = freq_list[(digital_f0 >>10) & 1023]
ch2_f0 = freq_list[(digital_f0 >>20) & 1023]
print "Selected channels are:\n\t{0} MHz\n\t{1} MHz\n\t{2} MHz".format(
        ch0_f0,ch1_f0,ch2_f0)

# update configuration
ch0_f0 = 700.0
ch1_f0 = 500.0
ch2_f0 = 900.0
digital_f0 = 0
digital_f0 = (abs(freq_list-ch2_f0).argmin()<<20) + \
        (abs(freq_list-ch1_f0).argmin()<<10) + \
        abs(freq_list-ch0_f0).argmin()
r2daq.write_int('r2daq_digital_f0',digital_f0)
```

3.5 Monitoring

The interface for monitoring will mostly be the same as for control, that is through python scripts as above and mostly built on calls to `read_int('...')`. Detailed monitoring will also include a mechanism for accessing the data stream, perhaps using `tcpdump` to capture network packets and analyze the data. For example, this could be used to implement a 'slow' spectrometer that may be useful to check IF power levels.

4 Data Capturing

Data is transmitted over UDP between the ROACH2 and the compute server. On the compute server the data is received, analyzed, and recorded to disk whenever a detection occurs.

The data processing on the compute server is built on the #| (hashpipe) framework. In this framework the data processing task is subdivided into separate components that each run in a different thread, as illustrated in Figure 1:

1. Thread T_r receives data over the network and loads it into the buffer B_{r-a} shared between T_r and T_a.

2. Thread T_a stores data that potentially needs to be recorded in a circular buffer $b_{a,c}$, and move simultaneous data for analysis into a buffer $b_{a,g}$ on the GPU. It then makes a call to CUDA code that performs an analysis on the data in $b_{a,g}$ to determine whether a trigger event has occurred. If such an event has occurred, a time-range is identified over which data is to be recorded. The corresponding data in $b_{g,c}$ is then loaded into the buffer B_{a-s} shared between T_a and T_s. This data is packetized with metadata that distinguishes packets pertaining to a new event from packets pertaining to continuation of an event-in-progress.

3. Thread T_s stores all the data that it receives to disk. Metadata received along with each unit of data is used to determine whether a new dataset needs to be created, or when data is a continuation of an existing data set.

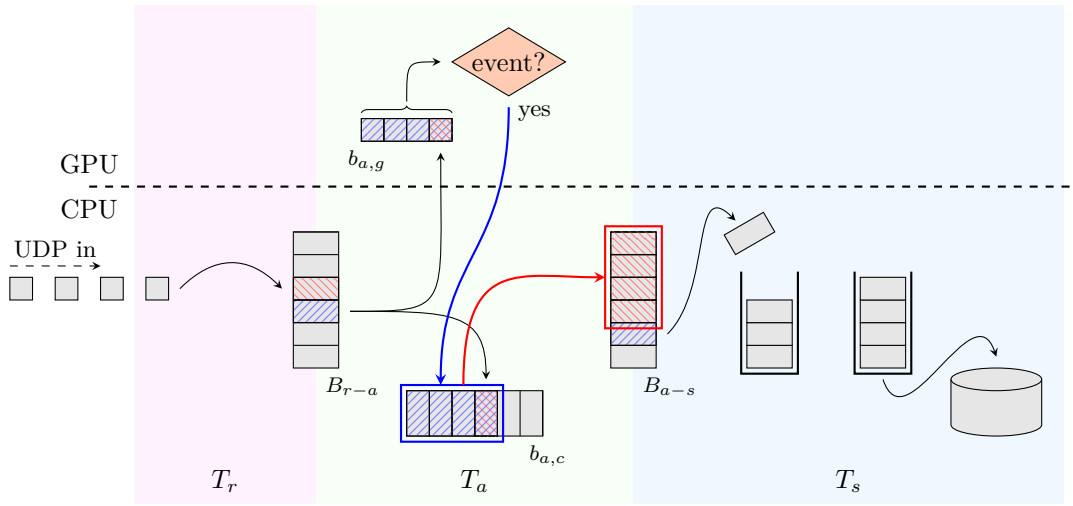

Figure 1: Pipelined data processing.

4.1 Receiver Thread

A code skeleton of the receiver thread T_r is shown in Listing 5 below.

In the main loop of this thread it will receive data from a socket connection, do some processing on it, and then write it to a shared buffer from where the next thread (analysis thread) in the pipeline will access it. Hashpipe manages thread-safe access to the shared buffers. The buffer structure is application dependent (lines 7–11), but the first field should be a hashpipe-style header. A method is also defined that creates the buffer (lines 13–26) according to the application requirements. The buffer will hold `num_data_units` units that are each `data_unit_size` in size.

When data is ready to be inserted into the output buffer, the thread first waits on a position to become available (lines 42–52), then copies the data (line 54), and then marks that position in the buffer filled (lines 56–58). At this point the next thread (analysis thread) is able to access the newly written data in the shared buffer, and the receiver thread can continue to receive more UDP data.

Each thread to be run on hashpipe is compiled as a shared library. When instantiating hashpipe it takes the names of threads to be pipelined as input arguments. Hashpipe then loads the corresponding libraries, initializes the shared buffers, and runs each thread's `run_method`. Upon loading each library should register itself as an available thread (lines 67–80), setting attributes such as its run-method and the shared input and output buffers it uses.

Listing 5: Outline of receiver thread.

```
/* include required libraries */

// hashpipe headers
#include "hashpipe.h"
#include "hashpipe_databuf.h"

// define output data buffer structure
typedef struct thread_r_out_databuf {
  hashpipe_databuf_t header;
  /* application specific fields here */
} trod_t;

// define method to create output data buffer
hashpipe_databuf_t *trod_create(int instance_id, int databuf_id) {
  size_t header_size = sizeof(hashpipe_databuf_t);

  // application specific requirements
  size_t data_unit_size;
  int num_data_units;

  /* set appropriate sizes */

  hashpipe_databuf_t *d = hashpipe_databuf_create(instance_id, databuf_id,
    header_size, data_unit_size, num_data_units);

}

// thread main
static void *run_method(hashpipe_thread_args_t * args) {
  int rv, index_out;
  trod_t *db_out = (trod_t *)args->obuf;

  /* do initialization here, e.g. open socket */

  // main loop
  while (run_threads()) {

    /* receive udp */

    /* do something with packetized data */

    // wait for available space in output buffer (trod_t)
    while ((rv = hashpipe_databuf_wait_free(
      (hashpipe_databuf_t *)db_out, index_out)) != HASHPIPE_OK) {
      if (rv == HASHPIPE_TIMEOUT) {
        aphids_log(&aphids_ctx, APHIDS_LOG_ERROR, "hashpipe databuf timeout");
        continue;
      } else {
        hashpipe_error(__FUNCTION__, "error waiting for free databuf");
        break;
      }
    }

    /* insert data in buffer */

```

```
56      // signal data ready in buffer and update index
57      hashpipe_databuf_set_filled((hashpipe_databuf_t *)db_out, index_out);
58      index_out = (index_out + 1) % db_out->header.n_block;
59
60    }
61
62    /* do clean-up */
63
64    return NULL;
65  }
66
67  // register thread with hashpipe
68  static hashpipe_thread_desc_t receiver_thread = {
69    name: "thread-r",
70    skey: "THREADR",
71    init: NULL,
72    run:  run_method,
73    ibuf_desc: {NULL},
74    obuf_desc: {trod_create}
75  };
76
77  static __attribute__((constructor)) void ctor()
78  {
79    register_hashpipe_thread(&receiver_thread);
80  }
```

4.2 Analysis Thread

A code skeleton of the analysis thread T_a is shown in Listing 6 below.

In the main loop of this thread it will receive data from the buffer shared with the receiver thread, do some analysis on it, and potentially write it to a shared buffer from where the next thread (storage thread) in the pipeline will access it. The input shared buffer `trod_t` is already defined in the receiver thread code, and it is more convenient to define the output shared buffer `tsid_t` in the storage thread code, so that here it is only necessary to reference these definitions (line 7).

The analysis thread will wait for data to become available in the input buffer (lines 20–34), copy data to local memory, e.g. circular buffer and buffer on GPU (line 36), and then mark the corresponding position in the buffer as free. Next some analysis is done on the data (line 42), and if needed data is written to the output buffer (lines 44–66). Some additional management may be needed, e.g. updating local buffer indexes (line 68).

When registering itself (lines 77–89) the analysis thread points to elsewhere defined methods `trod_create` and `tsid_create` for creating its input and output shared buffers, respectively. Hashpipe knows to create only a single instance of each shared buffer.

Listing 6: Outline of analysis thread.

```
1   /* include required libraries */
2
3   // hashpipe headers
4   #include "hashpipe.h"
5   #include "hashpipe_databuf.h"
6
7   /* include definition of trod_t and tsid_t */
8
9   // thread main
10  static void *run_method(hashpipe_thread_args_t * args) {
```

```
int rv, index_in, index_out, event;
trod_t *db_in = (trod_t *)args->ibuf;
tsid_t *db_out = (tsid_t *)args->obuf;

/* do initialization here, e.g. allocate buffers */

// main loop
while (run_threads()) {

  // read from the input buffer (trod_t)
  while ((rv = hashpipe_databuf_wait_filled(
    (hashpipe_databuf_t *)db_in, index_in)) != HASHPIPE_OK) {
    if (rv == HASHPIPE_TIMEOUT) {
      // need to check run_threads here again
      if (run_threads())
        continue;
      else
        break;
    } else {
      // raise an error and exit thread
      hashpipe_error(__FUNCTION__, "error waiting for filled databuf");
      break;
    }
  }

  /* process data, e.g. copy to local buffer, copy to GPU */

  // release input buffer and update index
  hashpipe_databuf_set_free((hashpipe_databuf_t *)db_in, index_in);
  index_in = (index_in + 1) % db_in->header.n_block;

  /* do analysis, event detection, etc. */

  if (event) {

    /* identify data to record, packetize */

    // wait for available space in output buffer (tsid_t)
    while ((rv = hashpipe_databuf_wait_free(
      (hashpipe_databuf_t *)db_out, index_out)) != HASHPIPE_OK) {
      if (rv == HASHPIPE_TIMEOUT) {
        aphids_log(&aphids_ctx, APHIDS_LOG_ERROR, "hashpipe databuf timeout");
        continue;
      } else {
        hashpipe_error(__FUNCTION__, "error waiting for free databuf");
        break;
      }
    }

    /* insert data-to-record in buffer */

    // signal data ready in buffer and update index
    hashpipe_databuf_set_filled((hashpipe_databuf_t *)db_out, index_out);
    index_out = (index_out + 1) % db_out->header.n_block;

  }
```

```
67
68      /* do managing */
69
70    }
71
72    /* some cleanup code here */
73
74    return NULL;
75  }
76
77  // register thread with hashpipe
78  static hashpipe_thread_desc_t analysis_thread = {
79    name: "thread-a",
80    skey: "THREADA",
81    init: NULL,
82    run:  run_method,
83    ibuf_desc: {trod_create},
84    obuf_desc: {tsid_create}
85  };
86
87  static __attribute__((constructor)) void ctor() {
88    register_hashpipe_thread(&analysis_thread);
89  }
```

4.3 Storage Thread

A code skeleton of the storage thread T_s is shown in Listing 7 below.

In the main loop of this thread it will receive data from the buffer shared with the analysis thread, and write it to disk. Most of the content in this listing is similar to that which has appeared for the other threads. The definition of the input buffer `tsid_t` as referenced in the analysis thread appears here (lines 7–11) as well as the creation method (lines 13–26). When this thread reads data from that buffer it first makes a local copy (line 54) in order to release the buffer space (lines 56–58) and then performs the more time-consuming tasks such as writing to disk (line 60).

Listing 7: Outline of storage thread.

```
1   /* include required libraries */
2
3   // hashpipe headers
4   #include "hashpipe.h"
5   #include "hashpipe_databuf.h"
6
7   // define input data buffer structure
8   typedef struct thread_s_in_databuf {
9     hashpipe_databuf_t header;
10    /* application specific fields here */
11  } tsid_t;
12
13  // define method to create input data buffer
14  hashpipe_databuf_t *tsid_create(int instance_id, int databuf_id) {
15    size_t header_size = sizeof(hashpipe_databuf_t);
16
17    // application specific requirements
18    size_t data_unit_size;
19    int num_data_units;
20
```

```
21    /* set appropriate sizes */
22
23    hashpipe_databuf_t *d = hashpipe_databuf_create(instance_id, databuf_id,
24      header_size, data_unit_size, num_data_units);
25
26  }
27
28  // thread main
29  static void *run_method(hashpipe_thread_args_t * args) {
30    int rv, index_in;
31    tsid_t *db_in = (tsid_t *)args->ibuf;
32
33    /* do initialization here */
34
35    // main loop
36    while (run_threads()) {
37
38      // read from the input buffer (tsid_t)
39      while ((rv = hashpipe_databuf_wait_filled(
40        (hashpipe_databuf_t *)db_in, index_in)) != HASHPIPE_OK) {
41        if (rv == HASHPIPE_TIMEOUT) {
42          // need to check run_threads here again
43          if (run_threads())
44            continue;
45          else
46            break;
47        } else {
48          // raise an error and exit thread
49          hashpipe_error(__FUNCTION__, "error waiting for filled databuf");
50          break;
51        }
52      }
53
54      /* process data, e.g. local copy */
55
56      // release input buffer and update index
57      hashpipe_databuf_set_free((hashpipe_databuf_t *)db_in, index_in);
58      index_in = (index_in + 1) % db_in->header.n_block;
59
60      /* further process, e.g. create files, write to disk */
61
62    }
63
64    /* do clean-up */
65
66    return NULL;
67  }
68
69  // register thread with hashpipe
70  static hashpipe_thread_desc_t storage_thread = {
71    name: "thread-s",
72    skey: "THREADS",
73    init: NULL,
74    run:  run_method,
75    ibuf_desc: {tsid_create},
76    obuf_desc: {NULL}
```

```
77 };
78
79 static __attribute__((constructor)) void ctor()
80 {
81   register_hashpipe_thread(&storage_thread);
82 }
```