Hashpipe Structures

Deepthi Gorthi

April 19, 2018

This file is a compilation of all the handy structures and functions Hashpipe provides for building a fast data capture and recording pipeline.

1 Thread Structures and Functions

1.1 `hashpipe_thread_desc_t`

```
struct hashpipe_thread_desc {
  const char * name;
  const char * skey;
  initfunc_t init;
  runfunc_t run;
  databuf_desc_t ibuf_desc;
  databuf_desc_t obuf_desc;
};
```

The `hashpipe_thread_desc` structure is used to store metadata describing a hashpipe thread. Typically a hashpipe plugin will define one of these hashpipe thread descriptors per hashpipe thread.

1. `const char *name`: String containing the thread name. Hashpipe threads are identified by their names which need to be registered with `register_hashpipe_thread()`. This is used to match command line thread specifiers to thread metadata so that the pipeline can be constructed as specified on the command line.

2. `const char *skey`: String containing the thread's status buffer "status" key. It is typically 8 characters or less, uppercase and ends with

"STAT". If it is `non-NULL` and non-empty, `HASHPIPE` will automatically store/update this key in the status buffer with the thread's status at initialization ("init") and exit ("exit").

3. `initfunc_t init`: Pointer to thread's initialization function. The thread initialization function can be null if no special initialization is needed. If provided, it must point to a function with the following signature:

   ```
   int my_thread_init_funtion(hashpipe_thread_args_t *args)
   ```

4. `runfunc_t run`: Pointer to thread's run function. The thread run function must have the following signature:

   ```
   void my_thread_run_funtion(hashpipe_thread_args_t *args)
   ```

5. `ibuf`: Structure describing thread's input data buffer (if any)

6. `obuf`: Structure describing thread's output data buffer (if any). The data buffer description structure used for ibuf and obuf currently contains one function pointer:

   create - A pointer to a function that creates the data buffer.

   Future `HASHPIPE` versions may introduce additional data buffer fields. `ibuf.create` should be `NULL` for input-only threads and `obuf.create` should be `NULL` for output-only threads. Having both `ibuf.create` and `obuf.create` set to `NULL` is invalid and the thread will not be used. The create function must have the following signature:

   ```
   hashpipe_databuf_t *my_create_function(
        int instance_id, int databuf_id)
   ```

1.2 hashpipe_thread_args

This structure passed (via a pointer) to the application's thread initialization and run functions. The 'user_data' field can be used to pass info from the init function to the run function.

1. `hashpipe_thread_desc_t *thread_desc`

2. `int instance_id`

3. `int input_buffer`
4. `int output_buffer`
5. `unsigned int cpu_mask`: 0 means use inherited
6. `int finished`
7. `pthread_cond_t finished_c`
8. `pthread_mutex_t finished_m`
9. `hashpipe_status_t st`
10. `hashpipe_databuf_t *ibuf`
11. `hashpipe_databuf_t *obuf`
12. `void *user_data`

1.3 Useful Functions

1. `int run_threads()`: Function threads used to determine whether to keep running.
2. `register_hashpipe_thread (hashpipe_thread_desc_t *ptm)`: Function should be used by pipeline plugins to register threads with the pipeline executable.
3. `hashpipe_thread_desc_t *find_hashpipe_thread(char *name)`: This function can be used to find hashpipe threads by name. It is generally used only by the hashpipe executable. Returns a pointer to its `hashpipe_thread_desc_t` structure or `NULL` if a test with the given name is not found. Names are case sensitive.
4. `void list_hashpipe_threads(FILE *f)`: List all known hashpipe threads to file pointed to by the file pointer.
5. `unsigned int get_cpu_affinity()`: Get the CPU affinity of calling thread.

2 Data Buffer Structures and Functions

2.1 `hashpipe_databuf_t`

1. `char data_type[64]`: Type of data in the buffer
2. `size_t header_size`: Size of each block header in bytes
3. `size_t block_size`: Size of each data block in bytes.
4. `int n_block`: Number of data blocks in buffer
5. `int shmid`: ID of this shared memory segment
6. `int semid`: ID of locking semaphore set

2.2 Associated functions

1. **Key**:

   `key_t hashpipe_databuf_key()`

   Get the base key to use for **all** hashpipe databufs. The base key is obtained by calling the ftok function, using the value of `$HASHPIPE_KEYFILE`, if defined, or `$HOME` from the environment or, if `$HOME` is not defined, by using `tmp`. By default (i.e. no `HASHPIPE_KEYFILE` in the environment), this will create and connect to a user specific set of shared memory buffers (provided `$HOME` exists in the environment), but if desired users can connect to any other set of memory buffers by setting `HASHPIPE_KEYFILE` appropriately.

2. **Create Databuf**:

   ```
   hashpipe_databuf_t *hashpipe_databuf_create(int
      instance_id,int databuf_id, size_t
      header_size, size_t block_size, int n_block)
   ```

   Create a new shared mem area with given params. Returns pointer to the new area on success, or `NULL` on error. Returns error if an existing shmem area exists with the given shmid and different sizing parameters.

3. **Get Databuf**:

   ```
   hashpipe_databuf_t *hashpipe_databuf_attach(int instance_id
   , int databuf_id)
   ```

   Return a pointer to a existing shmem segment with given id. Returns error if segment does not exist

4. **Detach Databuf**:

   ```
   int hashpipe_databuf_detach(hashpipe_databuf_t *d)
   ```

   Detach from shared mem segment

5. **Clear Databuf**:

   ```
   void hashpipe_databuf_clear(hashpipe_databuf_t *d)
   ```

6. **Reset Pointer location**:

   ```
   char *hashpipe_databuf_data(
      hashpipe_databuf_t *d, int block_id)
   ```

   Returns pointer to the beginning of the given data block.

7. **Get lock status**:

   ```
   int hashpipe_databuf_block_status(
      hashpipe_databuf_t *d, int block_id)

   int hashpipe_databuf_total_status(
      hashpipe_databuf_t *d)

   uint64_t hashpipe_databuf_total_mask(
      hashpipe_databuf_t *d)
   ```

   Returns lock status for given block_id, or total for whole array.

8. **Locking functions**:

   ```
   int hashpipe_databuf_wait_filled(
      hashpipe_databuf_t *d, int block_id)

   int hashpipe_databuf_busywait_filled(
      hashpipe_databuf_t *d, int block_id)
   ```

```
int hashpipe_databuf_set_filled(
   hashpipe_databuf_t *d, int block_id)

int hashpipe_databuf_wait_free(
   hashpipe_databuf_t *d, int block_id)

int hashpipe_databuf_busywait_free(
   hashpipe_databuf_t *d, int block_id)

int hashpipe_databuf_set_free(
   hashpipe_databuf_t *d, int block_id)
```

Databuf locking functions. Each block in the buffer can be marked as free or filled. The "wait" functions block (i.e. sleep) until the specified state happens. The "busywait" functions busy-wait (i.e. do NOT sleep) until the specified state happens. The "set" functions put the buffer in the specified state, returning error if it is already in that state.

3 Status Handling

Routines dealing with the hashpipe status shared memory segment. Info is passed through this segment using a FITS-like keyword=value syntax.

```
typedef struct {
    int instance_id;
    int shmid;
    sem_t *lock;
    char *buf;
} hashpipe_status_t;
```

Status handling functions

1. Semaphore name: Stores the hashpipe status (POSIX) semaphore name in semid buffer of length size. Returns 0 (no error) if semaphore name fit in given size, returns 1 if semaphore name is truncated.

   The hashpipe status semaphore name is `$HASHPIPE_STATUS_SEMNAME` (if defined in the environment) or `$HASHPIPE_KEYFILE_hashpipe_status`

(if defined in the environment) or `$HOME_hashpipe_status` (if defined in the environment) or `"/tmp_hashpipe_status"` (global fallback). Any slashes after the leading slash are converted to underscores.

```
int hashpipe_status_semname(int instance_id, char * semid,
size_t size)
```

2. Instance ID number:

   Returns non-zero if the status buffer for instance_id already exists.

   ```
   int hashpipe_status_exists(int instance_id)
   ```

3. Attach buffer:

   Return a pointer to the status shared mem area, creating it if it doesn't exist. Attaches/creates lock semaphore as well. Returns nonzero on error.

   ```
   int hashpipe_status_attach(int instance_id, hashpipe_status_t
    *s)
   ```

4. Detach buffer: Detach from shared mem segment

   ```
   int hashpipe_status_detach(hashpipe_status_t *s)
   ```

5. Locking/Unlocking status buffer:

   Lock/unlock the status buffer. hashpipe_status_lock() will sleep while waiting for the buffer to become unlocked. hashpipe_status_lock_busywait will busy-wait while waiting for the buffer to become unlocked. Return non-zero on errors.

   ```
   int hashpipe_status_lock(hashpipe_status_t *s)

   int hashpipe_status_lock_busywait(
      hashpipe_status_t *s)

   int hashpipe_status_unlock(hashpipe_status_t *s)
   ```

6. Check formatting:

   Check the buffer for appropriate formatting (existence of `"END"`). If not found, zero it out and add `END`.

```
void hashpipe_status_chkinit(hashpipe_status_t *s)
```

7. Clear: Clear out whole buffer

   ```
   void hashpipe_status_clear(hashpipe_status_t *s)
   ```

8. Thread safe lock/unlock:

   Thread-safe lock/unlock macros for status buffer used to ensure that the status buffer is not left in a locked state. Each hashpipe_status_lock_safe or hashpipe_status_lock_busywait_safe must be paired with a hashpipe_status_unlock_safe in the same function and at the same lexical nesting level. See "man pthread_cleanup_push" for more details.

   Note This header file does not include pthread.h where pthread_cleanup_push and pthread_cleanup_pop are defined. Users of the macros defined here must explicitly include pthread.h themselves.

```
#define hashpipe_status_lock_safe(s) \
    pthread_cleanup_push((void (*)(void *))
  hashpipe_status_unlock, s); \
    hashpipe_status_lock(s);

#define hashpipe_status_lock_busywait_safe(s) \
    pthread_cleanup_push((void (*)(void *))
  hashpipe_status_unlock, s); \
    hashpipe_status_lock_busywait(s);

#define hashpipe_status_unlock_safe(s) \
    hashpipe_status_unlock(s); \
    pthread_cleanup_pop(0);
```

4 Socket and UDP packet handling

4.1 "hashpipe_pktsock.h"

Routines for dealing with packet sockets. See `man 7 packet` and `packet_mmap.txt` for more info.

```
struct hashpipe_pktsock {
  unsigned int frame_size;
  unsigned int nframes;
  unsigned int nblocks;
  int fd;
  unsigned char *p_ring;
  int next_idx;
};
```

4.1.1 Functional preprocessors

1. **Header**: Return header field 'h' from packet frame pointed to by 'p'

   ```
   TPACKET_HDR(p,h) (((struct tpacket_hdr *)p)->h)
   ```

2. **MAC**: Return pointer to MAC header inside packet frame pointed to by 'p'.

   ```
   PKT_MAC(p) (p+TPACKET_HDR(p, tp_mac))
   ```

3. **Network**: Return pointer to network (e.g. IP) packet inside packet frame pointed to by 'p'.

   ```
   PKT_NET(p) (p+TPACKET_HDR(p, tp_net))
   ```

4. **Check UDP**: Returns true (non-zero) if this is a UDP packet.

   ```
   PKT_IS_UDP(p) ((PKT_NET(p)[0x09]) == IPPROTO_UDP)
   ```

5. **Destination**: Returns UDP destination port of packet. **NB:** this assumes but does NOT verify that the packet is a UDP packet!

   ```
   PKT_UDP_DST(p) (((PKT_NET(p)[0x16]) << 8) |
                   ((PKT_NET(p)[0x17]) ))
   ```

6. **UDP Size**: Returns size of UDP packet (including the 8 byte UDP header). **NB:** this assumes but does NOT verify that the packet is a UDP packet!

```
PKT_UDP_SIZE(p) (((PKT_NET(p)[0x18]) << 8) |
                 ((PKT_NET(p)[0x19])     ))
```

7. **Get UDP Payload**: Returns pointer to UDP packet payload. **NB:** this assumes but does NOT verify that the packet is a UDP packet!

```
PKT_UDP_DATA(p) (PKT_NET(p) + 0x1c)
```

4.1.2 Useful Functions

1. Bind to Socket and attach it to a buffer:

   `p_ps` should point to a `struct pktsock` that has been initialized by caller with desired values for the sizing parmaters `frame_size`, `nframes`, and `nblocks`. `nblocks` MUST be a multiple of `nframes` and the resulting block size (i.e. `frame_size * nframes / nblocks`) MUST be a multiple of `PAGE_SIZE`.

   `ifname` should specify the name of the interface to bind to (e.g. "eth2").

   `ring_type` should be `PACKET_RX_RING` or `PACKET_TX_RING`.

   Returns 0 for success, non-zero for failure. On failure, errno will be set.

   Upon successful compltetion, `p_ps->fd` will the the file descriptor of the socket, and `p_ps->p_ring` will be a poitner to the start of the ring buffer.

```
int hashpipe_pktsock_open(struct
   hashpipe_pktsock *p_ps, const char *ifname,
   int ring_type)
```

2. Get frame:

   Return pointer to frame or NULL if no frame ready.

   If a non-NULL frame pointer is returned, the caller MUST release the frame back to the kernel (via `pktsock_release_frame`) once it is finished with the frame.

```
unsigned char *
   hashpipe_pktsock_recv_frame_nonblock(
struct hashpipe_pktsock *p_ps)

unsigned char * hashpipe_pktsock_recv_frame(
   struct hashpipe_pktsock *p_ps,
int timeout_ms)
```

3. Get matching frame:

   If no frame is ready, returns NULL. If a non-matching frame is ready, it is released back to the kernel and NULL is returned. Otherwise, returns a pointer to the matching frame.

   If a non-NULL frame pointer is returned, the caller MUST release the frame back to the kernel (via `pktsock_release_frame`) once it is finished with the frame.

   ```
   unsigned char *
      hashpipe_pktsock_recv_udp_frame_nonblock(
      struct hashpipe_pktsock *p_ps, int dst_port)
   ```

4. Wait for frame:

   If no frame is ready, wait up to `timeout_ms` to get a frame. If no frame is ready after timeout, returns NULL. If frame does not match, releases non-matching frame back to the kernel and returns NULL. Otherwise returns pointer to matching frame.

   If a non-NULL frame pointer is returned, the caller MUST release the frame back to the kernel (via `pktsock_release_frame`) once it is finished with the frame.

   ```
   unsigned char * hashpipe_pktsock_recv_udp_frame(
      struct hashpipe_pktsock *p_ps, int dst_port,
      int timeout_ms)
   ```

5. Release frame: Releases frame back to the kernel. The caller must be release each frame back to the kernel once the caller is done with the frame.

   ```
   void hashpipe_pktsock_release_frame(unsigned char * frame)
   ```

6. Get stats: Stores packet counter and drop counter values in `*p_pkts` and `*p_drops`, provided they are non-NULL. This makes it possible to request one but not the other.

```
void hashpipe_pktsock_stats(struct
   hashpipe_pktsock *p_ps,unsigned int *p_pkts,
   unsigned int *p_drops)
```

7. Close socket: Unmaps kernel ring buffer and closes socket.

```
int hashpipe_pktsock_close(struct hashpipe_pktsock *p_ps)
```

4.2 hashpipe_udp.h

```
struct hashpipe_udp_params {
    char sender[80];
    int port;
    char bindhost[80];
    int bindport;
    size_t packet_size;
    char packet_format[32];

    /* Derived from above: */
    int sock;
    struct addrinfo sender_addr;
    struct pollfd pfd;
};

struct hashpipe_udp_packet {
    size_t packet_size;  /* packet size, bytes */
    char data[HASHPIPE_MAX_PACKET_SIZE]
   __attribute__ ((aligned(128)));
};
```

Functions

1. Use sender and port fields in param struct to init the other values, bind socket, etc.

```
int hashpipe_udp_init(struct hashpipe_udp_params *p)
```

2. Close out socket, etc.

   ```
   int hashpipe_udp_close(struct hashpipe_udp_params *p)
   ```

5 Error Handling

5.1 Exit codes

HASHPIPE_OK	0	Everything is great
HASHPIPE_TIMEOUT	1	Call timed out
HASHPIPE_ERR_GEN	-1	Super non-informative
HASHPIPE_ERR_SYS	-2	Failed system call
HASHPIPE_ERR_PARAM	-3	Parameter out of range
HASHPIPE_ERR_KEY	-4	Requested key doesn't exist
HASHPIPE_ERR_PACKET	-5	Unexpected packet size

5.2 Error logging

1. log error: Call this to log an error message.

   ```
   void hashpipe_error(const char *name, const char *msg, ..)
   ```

2. log warning: Call this to log an warning message.

   ```
   void hashpipe_warn(const char *name, const char *msg, ...)
   ```

3. log info: Call this to log an informational message.

   ```
   void hashpipe_info(const char *name, const char *msg, ...)
   ```