**Client/Server interface for camera and spectrometer (and others later)**

I redefined an interface that should give us more flexibility and hopefully will be more compatible with Python or whatever else we plan to use.

Communication is over a persistent socket opened on monitored ports. Expect each “tool” to have its own port even if they are physically on the same computer.

**DCx Camera:** port 985

Once opened, communication is always a request/response sequence. The request and response use a common structure containing the action and minimal information as necessary.

#pragma pack(4)

typedef struct \_DCX\_MSG {

uint32\_t msg;

uint32\_t msgid;

uint32\_t rc;

uint32\_t data\_len;

} DCX\_MSG;

#pragma pack()

All parameters are 32 bit integers and packed in network order (ntohl and htonl).

For **requests**, the parameters are:

* **msg**: command or action to be executed. For the DCx camera, allowed values are:

#define DCX\_QUERY\_VERSION (1) /\* Return version of the server code \*/

#define DCX\_GET\_CAMERA\_INFO (2) /\* Return structure with camera data \*/

#define DCX\_ACQUIRE\_IMAGE (3) /\* Acquire image only (local storage) \*/

#define DCX\_GET\_IMAGE\_INFO (4) /\* Return info on the image \*/

#define DCX\_GET\_IMAGE\_DATA (5) /\* Transfer the actual image data \*/

* **msgid**: identifier associated with the command. Will be used if we implement queuing, but currently ignored.
* **rc**: generally, not used but may be used to pass a single parameter. Should be zero normally.
* **data\_len**: length (in bytes) of associated data to be sent, or zero if unneeded. If there is additional data, this informs the server of the number of bytes that should be expected in the subsequent (immediate) send(). Currently no commands transmit additional data.

The **response** to a request will have the same structure, with parameters

* **msg**: if no error, will be same as the request msg. If it does not match, consider as a fatal error.
* **msgid**: returned identifier associated with the command (for queued requests eventually)
* **rc**: return code from command. 0 if successful except for DCX\_QUERY\_VERSION (then version #)
* **data\_len**: length (in bytes) of associated data that will immediately follow. Client should allocate a buffer and recv() the data sent. Interpretation of the data depends on the command.

The request/response are in network order, but the no translation is used by default for the additional data sent either as part of the request or as part of the response. Managing interpretation of the byte order is left to the program.

**DCx CAMERA ACTIONS:**

**DCX\_QUERY\_VERSION:** No parameters. Returns version number embedded in the server code (1001 currently). It is important to verify the version number corresponds as this API may change rapidly.

**DCX\_GET\_CAMERA\_INFO:** No parameters. Returns a structure (in the second send()) containing details about the camera. The structure is defined as

#pragma pack(4)

typedef struct \_DCX\_STATUS {

char manufacturer[32]; /\* Camera manufacturer \*/

char model[32]; /\* Camera model \*/

char serial[32]; /\* Camera serial number \*/

char version[32]; /\* Camera version \*/

char date[32]; /\* Firmware date \*/

uint32\_t CameraID; /\* Camera ID (as set in EEPROM) \*/

DCX\_IMAGE\_TYPE color\_mode; /\* Monochrome or color mode \*/

uint32\_t pixel\_pitch; /\* Pixel pitch in um \*/

double fps; /\* Frame rate (frames per second) \*/

double exposure; /\* Current exposure (ms) \*/

double gamma; /\* Gamma value \*/

uint32\_t master\_gain; /\* Gains in non-linear range [0,100] \*/

uint32\_t red\_gain, green\_gain, blue\_gain;

uint32\_t color\_correction; /\* 0,1,2,4,8 ==> disable, enable, BG40, HQ, IR Auto \*/

double color\_correction\_factor;

} DCX\_STATUS;

#pragma pack()

**DCX\_ACQUIRE\_IMAGE:** No parameters. Acquire an image using the current acquisition parameters. The image is stored on the server side for subsequent readout, along with details of the acquisition.

**DCX\_GET\_IMAGE\_INFO:** No parameters. Returns a structure describing the acquisition parameters. The data is persistent on the server until a subsequent DCX\_ACQUIRE\_IMAGE command. Note that the memory\_pitch element is critical; this is the byte spacing between subsequent rows of the image, with potential for padding at the end of each line for memory alignment.

#pragma pack(4)

typedef struct \_DCX\_IMAGE\_INFO {

uint32\_t width, height; /\* Image width and height \*/

uint32\_t memory\_pitch; /\* Bytes between each row (allocate pitch\*height) \*/

double exposure; /\* Current exposure (ms) \*/

double gamma; /\* Gamma value \*/

uint32\_t master\_gain; /\* Gains in non-linear range [0,100] \*/

uint32\_t red\_gain, green\_gain, blue\_gain;

uint32\_t color\_correction; /\* 0,1,2,4,8 ==> disable, enable, BG40, HQ, IR Auto \*/

double color\_correction\_factor;

uint32\_t red\_saturate, green\_saturate, blue\_saturate; /\* Number saturated pixels \*/

} DCX\_IMAGE\_INFO;

#pragma pack()

**DCX\_GET\_IMAGE\_DATA:** No parameters. Returns a very large second send() containing the entire image (typically 3.93 MB for color images and 1.31 MB for monochrome). For color images, the sequence is BGR in memory.

I implemented a “test” server that does not query a physical camera but responds to all of these messages. It compiles and runs under Windows, and might work okay under Linux. There is a corresponding client that can access this test server, or access the actual camera in the lab (change the IP address of the server).