Camera Control Code Documentation

Kenny Young

August 7, 2013

Contents

1 Introduction

This Document is intended to outline the structure of the code for controlling the SNO+ camera system. This includes:

1. the C++ code which is called to preform the actual image capture

2. the python script which calls this code and also takes care of database upload

3. the "couchapp" which is uploaded to the same database as the images and allows online viewing of image capture runs

I'll discuss each of these components individually.

2 C++ image capture code

This is the C++ code responsibly for actually communicating with the Cameras and taking pictures. The code uses the open source library "libgphoto2" for camera control and also borrows some code from it's front end "gphoto2" where convenient. It also uses openMP to allow parallel image capture among all attached cameras. The code is spread over the following files:

- capture.cxx
- gp-params.cxx
- logger.cxx
- funcs.cxx
- actions.cxx

Each of these files will be explained below.

2.1 capture.cxx

The first file, capture.cxx is essentially the main function. It contains the function "capture()" which is called in order to take a picture. Currently capture takes one argument "runnumber" which should either be 0 for a rope-net capture run or the run-number of the corresponding calibration run. Passing this argument to capture is only necessary so that it can be included in the directory of the captured images when they are saved to disk. This function returns a C++ struct, "captureReturn", containing 4 values. The first return value is an errorFlag (equal to 0 if no error occurred, or a non-zero error code otherwise). The next two values are "captureTime" which is the time-stamp for the start of the capture run, and activeCams which is the number of attached cameras detected. These two values are passed in order to be uploaded to the database along with the images. The last value is captureFolder which is the directory of the folder containing the

images and logfile for the capture-run, it is passed so that the python script knows where to find the images and logfile to upload to the database.

The capture.cxx file also contains a few other helper functions which I'll outline briefly. First list_detect_action is the function which is called to detect and initialize any attached cameras. This function takes as arguments an array of gp-params (a class which contains information about and functions to control a camera) and a "logger" which simply writes information about the run to a logfile. The return value is the number of cameras detected. Second ctx_error_func is a function which is passed to libgphoto2 and called by libgphoto2 in the event of a context error. This functions only purposed is to notify the capture-script of any context errors thrown by libgphoto2, write them to the logfile, and set an errorFlag appropriately. It uses direct string comparison to analyze the error received which is messy but seemingly the only way provided by libgphoto2 to determine the particular context error which occurred. The only other function in the file, sig_handler, serves only to print a message and delete the lockfile if the program receives any exit signals, if all goes well this should never be called.

The flow of execution for the function capture() can be outlined as follows:

1. check for lockfile and return 7 if found, otherwise create lockfile to prevent multiple instances of program running at once
2. Create directory within /captures to store captured images and logfile for the run, currently directory is /captures/[runnumber]/[timestamp]
3. Create logfile in this directory
4. Call list_detect_action to detect and initialize attached cameras; info for detected cameras is now at the array pointer GPParams
5. Iterate over each detected camera, openMP is used to parallelize this iteration
6. Set internal time on each camera so that timestamp in image metadata is accurate
7. Tell each camera to capture an image (procedure for this is defined in gp_params class and will be detailed below)
8. Exit each camera
9. return error-flag, time-stamp and camera-count to calling script

2.2 gp-params.cxx

The file gp-params.cxx defines a class which contains information about each camera and functions for controlling them on an individual level. It is a highly modified version of the file gp-params.c taken from gphoto2. The data members of this class are as follows:

- Camera *fCamera: an instance of the Camera object defined in libgphoto2, acts as a reference for the camera when telling libgphoto2 functions which camera to communicate with
- string camID: the identifier for the camera in the context of the SNO+ experiment (i.e. Camera1,2,3,4,5,P)
- int fSerialNumber: serial number of the camera (this is used to determine camID from a hardcoded list)
- GPContext *fContext: a libgphoto2 context, this allows passing of errors from libgphoto2 to the program
- errorFlag: this is set by the context error function param_ctx_error_func whenever a context error occurs, it is class member because this provides an easy way to pass the information back to the class
- string fFolder: this is the same as the captureFolder outlined above, it tells gp-params where to download captured images
- logger fLogFile: the "logger" for this run (same one created in the capture function above), allows writing of log information from within gp-params member functions
- int logNum: a number specifying the subsection of the logfile to write to (information for each camera is written sequentially to a different "subsection" instead of writing logged events chronologically; since the cameras capture in parallel this makes the logfile much more readable)

Each of these data members is set appropriately at various points in the execution of the capture() function.

There are also 4 function members:

- int capture_to_file(): instruct camera to capture an image to the directory specified by fFolder, image file name will be the camera ID followed by time stamp (e.g. Camera1_2013-07-26T13:09:25.nef)

- void save_captured_file(CameraFilePath camera_file_path): saves a captured image to disk, this is currently a private member function only called within capture_to_file()

- int set_config_int(char* option,int value): attempt to set the configuration named "option" to the integer value "value" int the camera options, currently this is only used to set the time on the cameras

- int exit_camera(): exit the camera (tells libgphoto2 to preform nessesary cleanup on exit) only necessary if multiple capture-runs are done in one powerup but can prevent some errors in this case

Each of these functions (except save_captured_file which is internal) are called from capture().

Other than these the file contains a simple constructor and destructor along with param_ctx_error_func which is essentially equivalent to the context error function in capture.cxx. It is convenient to define this separate context error function within the class for a number of reasons. This context error function writes errors to the logfile subsection corresponding to the given camera and sets the errorFlag member of its GPParams instance.

As a side note, if the camera is set to capture NEF and JPG (i.e. image quality=NEF(RAW)+JPEG) capture_to_file() will attempt to download both files. This is slightly non-trivial as calling the libgphoto2 function "gp_camera_capture" will by default only wait for one file to be received from the camera. After that code was added to wait for additional events from the camera. This code waits for at most 10 events (arbitrary number which seemed redundantly sufficient to catch an extra images) with a wait-time of 200ms. If the event type is "GP_EVENT_FILE_ADDED" the event should have corresponding data consisting of a file path which can then be used to download the new image. Otherwise if the event type is GP_CAPTURE_COMPLETE or GP_EVENT_TIMEOUT (thrown if no event is received from camera in 200ms) the program assumes it is ok to exit and breaks out of the loop.

2.3 logger.cxx

This file defines a simple class for logging information about a capture run. It has a constructor which takes one argument, the pathname for a logfile for the current run. It has a fstream member which allows writing to the specified logfile. One instance of logger is created by capture() for each run in the directory specified by captureFolder, the name of the logfile is "logfile_[timestamp].log", where timestamp is the start time for the run. The

logger has several few member functions used for writing to the associated file:

- void write(string line): write the given string to the next a line of the associated logfile, also places a timestamp at the start of the line which is the time elapsed since the start of the capture run
- void write(string line,int i): overloaded write method, writes the given string to *subsection* i of the logfile. This actually writes the text to a stringstream temporarily until the combine() function is called, at which point all subsections are written to the actual logfile.
- void split(int i): create i subsections of the logfile, each of which can be written to individually by calling write([string],[j]) where j=0..i-1
- void combine(): write out the i subsections of the logger to the actual logfile

All these methods are called from various points in capture() and also within gp-params with write() being called for all significant events in the capture run, and also for any errors that occur. The final result should look something like this:

```
0.000073:Start time for this run: Thu Jun 20 10:37:45 2013

0.000174:Detecting ports.
0.187349:Detecting camera drivers.
0.214206:Detecting supported cameras.
0.221129:2 supported cameras detected.
0.221152:1 Nikon DSC D5000 (PTP mode) usb:002,019
0.504413:2 Nikon DSC D5000 (PTP mode) usb:002,018


0.804779:Camera ID : 5062213(Camera4)
0.804788:Updating time on camera.
0.994027:Set time to:Thu Jun 20 10:37:46 2013.
0.994039:Capturing image:
0.994056:Capturing image to camera.
34.902427:File saved to camera as://capt0000.nef
34.902480:Saving file to disk as: captures/capture_40/Camera4.nef
34.902558:Retrieving file //capt0000.nef from camera.
34.925256:Deleting file //capt0000.nef from camera.
34.930592:Exiting camera.
```

```
0.804575:Camera ID : 5068091(Camera1)
0.804584:Updating time on camera.
0.989286:Set time to:Thu Jun 20 10:37:46 2013.
0.989299:Capturing image:
0.989316:Capturing image to camera.
34.899682:File saved to camera as://capt0001.nef
34.899760:Saving file to disk as: captures/capture_40/Camera1.nef
34.900000:Retrieving file //capt0001.nef from camera.
34.927738:Deleting file //capt0001.nef from camera.
34.930529:Exiting camera.
```

If any errors occurred those too would be listed at their time of occurence.

2.4 funcs.cxx

This file contains some simple functions that are used periodically throughout the code, they are defined here for lack of a better place and probably don't require much explanation, they are:

- string convertIntToString(int i): converts an integer to a string
- string makeTimestamp(time_t t): creates a human readable timestamp from a time_t value
- string serialToCamID(int serial): a switch for converting camera serial numbers to the associated ID within the SNO+ experiment

2.5 actions.cxx

This file is a highly reduced version of the gphoto2 actions.c file, it contains 2 functions copied essentially verbatim from gphoto2 which were convenient to use in this code. The functions are:

- GPPortInfoList* _get_portinfo_list (): detects usb (or other ports) this is used within list_detect_action()
- int _find_widget_by_name (GPParams *p, const char *name, CameraWidget **child, CameraWidget **rootconfig): searches the camera configuration tree for an option with a given name, used to find the "datetime" option for setting the time on the camera, could also be used if any other configuration of camera options was desired within this code

The above files are compiled to produce a C++ library libcapture.dylib which is wrapped by a python module using cython as detailed below. The function capture() is then called from a python script to capture images.

3 Python Script

In order to actually call the program a python script called capture_script.py is used. This script interfaces with the above detailed C++ code using cython. It also attempts to upload to the a couchdb database using the couchdb python module.

3.1 Cython C++ Wrapper

The script capture_script.py uses cython to interface with the above detailed C++ code and call capture(). As is easily seen by looking at the code in capture_script.py, the wrapper for capture() is "capture.py_capture()". The function capture.py_capture() is functionally the same as capture() listed above except that instead of a C struct it returns 3 python values directly. Like capture() capture.py_capture() takes runNumber as a single argument.

Using cython this python wrapping was extremely easy to accomplish, the relevant code is contained in "py_capture.pyx" which is easily seen to do no more than take the returned struct from the C++ function and parse it into the 3 python values which are returned. The contained "setup.py" file then tells cython to do all the heavy lifting. Upon running something like "python setup.py build_ext -i" (with cython installed) cython takes libcapture.dylib and produces a new C++ file py_capture.cpp and a python module "capture.so", which contains the py_capture() function that is actually called in "capture_script.py". The quoted command will build the module in place (i.e. in the working directory) which should make it accessible to capture_script.py in the same directory, if desired it should also be possible to build it into the standard location for installed python modules on the local machine.

3.2 Couchdb Database Upload

Aside from calling the C++ code to capture images the python script capture_script.py uses the couchdb python module to attempt to upload the resulting images to a couchdb database. For this to be successful there must be a running couchdb instance on the current machine which contains a database with a name matching the "dbName" variable. The function

couchdb.Server() will attempt to connect to the local couchdb and return a dictionary containing each database in the couchdb listed by name.

Alternatively the code can be modified to upload to a non-local couchdb by replacing the line couchdb.server() with couchdb.server([server url]), appropriate credentials (username/password) must be included by setting "couch.resource.credentials = ([username] , [password])", if required by the database. In this case it is not necessary to have couchdb running on the local machine (though the couchdb python module must still be installed). An example of this is commented out below the line "couch = couchdb.Server()" in capture_script.py.

The code to upload (non-image) information to the database takes the form of a dictionary (set of key, value pairs) given as an argument to db.save(). This creates a couchdb "document" in the database to which the images will be attached, information in this document can later be organized by key and viewed using the "couchapp" contained with the code, or by other methods. Uploading the actual images is slightly different as they must be uploaded as "attachments", it seems to be necessary to upload them 1 by 1. Uploading of the images, as well as the text from the logfile, is accomplished by a for loop which iterates over all files in the captureFolder (the directory in which the images are stored, returned by capture.py_capture()). The logic here is very simple, the loop checks the extension of each file in the directory, if it is .nef or .jpg it will attempt to upload the file as an image to the database. If on the other hand the extension is .log it will read the text from the file and attempt to upload the string to the database under the key "log".

3.3 Arguments

Currently the first argument to capture_script.py is runNumber which is supposed to be either 0 for rope-net capture runs or equal to the calibration run number for calibration runs. Depending on whether this value is non-zero or not the script while determine if the current run is rope-net or calibration and based on this decide what the remaining arguments mean. If the number of arguments does not match the required number of arguments for that run type (defined at the top of the script), or if any arguments do not have required type the script will return with error value 13, indicating bad parameters. The exact number and type of required parameters for each run type is **highly subject to change** as important information to have with the images is determined, the current arguments for each run type should be checked in the capture_script.py file.

3.4 Errors

Both the C++ code and python script include a number of error values that will be returned if the something goes wrong. It is hoped that these values will be helpful in solving any problems that may occur, however they are not necessarily guaranteed to be accurate or complete as unanticipated problems may arise. At the time of this writing a return value of 0 indicates a successful capture run, while error values from 1-10 indicate problems the running of the C++ capture code while error values 11+ indicate other problems (i.e. problems which occur in other parts of the python script). The error codes are as follows:

- 1-Lost communication with camera during run
- 2-Library issue (probably wrong version or installation problem)
- 3-No picture detected on a camera or camera unable to store picture (may indicate low voltage)
- 4-Unable to Save picture to disk (most likely full disk or permissions issue)
- 5-No Camera detected
- 6-Failed to set timestamp
- 7-Lockfile in place (indicating another instance of program is running)
- 8-Program out of memory
- 9-Failed to claim camera (probably indicating another process has claimed it or it is mounted on the filesystem)
- 10-Unknown image capture error (consult logfile for more information in this case)
- 11-Failed to connect to database for image upload
- 12-Unable to read logfile, text may have been corrupted, images may still have been uploaded without log
- 13-Invalid arguments passed to python script
- 14-Could not locate capture folder for database upload

4 Couchdb interface

All the files relating to the couchdb interface are contained in the directory couchdbinterface. The files in this directory are entirely unnecessary for the rest of the program to function and are instead related to creating a webinterface, in the form of a couchdb "design document", to the database after images are uploaded. In order to upload these files to the database "couchapp" should be installed on the machine (along with a running couchdb). I'll briefly discuss the most relevant files contained in this couchapp, for more information consult the couchdb and/or couchapp documentation.

First off the lib directory contains any javascript libraries to be used in the couchapp.

The views directory contains couchdb view functions, these functions run on each document in the database and in doing so create a data table ordered by a chosen key. Their output is generally given to a list function to create formatted output. Currently there is only one view function "by_timestamp" which outputs a table ordered by timestamp containing all the information in the documents.

The lists directory contains couchdb list functions, these functions iterate over the rows of the table output by and using a template output information from the documents in a specified form. Currently there is a list for all capture runs, a list for rope-net captures and a list for calibration captures each of which outputs a table with some information about available documents of the given type and links to pages displaying the images for each run. There is also a "mostRecent" list which simply redirects to a page displaying the most recent capture. Each of these lists uses the same view function (by_timestamp).

The shows directory contains couchdb show functions. these functions are passed a single specified document and using a template output it's information in a specified human-readable way. There is currently only one show function which produces a page displaying each of the images captured on a given run, as well as the text of the logfile for that run, this is intended as a quick verification that the capture was successful.

The templates directory contains the html templates used in the above listed functions, the templates use the very simple "mustache" template language to turn the raw data output from the above functions into an html webpage. The _attachments directory currently contains only a simple html index page which allows access links to the other available pages of the couchapp.

The whole interface is currently very raw and minimalistic, more work may be done to make it aesthetic and add additional functionality such as some method of downloading one or more capture runs which will have to

be determined.

5 installation

Modification of the contained Makefile will probably be nessesary to install this code.

To install this code first install libgphoto2 version 2.5.2 (it should run with any 2.5.x but with potentially decreased error handling, using 2.4.x and lower will likely cause run time errors, for this reason code has been added to check for versions other than 2.5.x and return an error upon running in this case). Please also ensure that libphoto2 is linked against libusb 0.x rather then libusb 1.x as the later may cause runtime errors or significant delays. Use of capture_script.py also requires Cython.

With the necessary packages installed change the library directories in the makefile to match the installation location on the local machine and then run make. This should produce a library libcapture.dylib and a python module called py_capture.so which is used by capture_script.py to call the C++ library to capture an image and then attempts to upload to a database. To install on linux libcapture.dylib in the makefile should be changed to libcapture.so. Currently the python module py_capture is built in place by running the makefile, if desired this should be easy to modify to instead build it to the standard location for python modules on the local machine.

In order for database upload to work it is necessary to install "couchdb", have it running, and create an appropriately named database (the database name can be set in "capture_scipt.py").

Additionally this directory should contain a file called couchdbinterface. This is not needed for the installation or running of the capture script or for database upload. It contains a "couchapp" which can be loaded to the database and defines an interface for viewing and possibly downloading the captured images. To install this couchapp first install "couchdb" and "couchapp" and then cd into couchdbinterface and with couchdb running, and the appropriate database created, do "couchapp push" (see couchapp documentation for details of the push command). Running "couchapp push" alone will install to a default database with default credentials which can be configured in "couchdbinterface/.couchapp.rpc".

6 running

In order to run the executable successfully the camera should not be mounted on the machine or claimed by another process. To run on mac you can use the shell-script found here "https://github.com/mejedi/mac-gphoto-enabler", to reversably disable some processes that tend to interfere. Alternatively to run once type "KILLALL PTPCamera" after powering the camera but before running ./capture_script.py, however this must be done each time the camera is powered down and then up again.

The Python Script ./capture_script.py may take a variable number of arguments depending on the type of the current run. A run is classified as either rope-net or calibration type. The first argument in either case is the run-number, if it is 0 the script interprets the run as a rope-net capture, otherwise it should be an integer equal to the run-number of the current calibration run. If the wrong number or wrong type of arguments are entered the script will return with error code 13 and do nothing else. As stated earlier the arguments for a given type of run are **highly subject to change**, please look in capture_script.py to see the current required arguments.